AD-A047 475 HONEYWELL INC MINNEAPOLIS MINN SYSTEMS AND RESEARCH --ETC F/G 9/2
DISTRIBUTED DATA PROCESSING TECHNOLOGY. VOLUME IV. APPLICATION --ETC(U)
SEP 77 M G GOUDA , Y W HAN, E D JENSEN DASG60-76-C-0087
UNCLASSIFIED 77SRC68 NL

1 OF 2
AD
A047475

DISTRIBUTED DATA PROCESSING TECHNOLOGY

DASG60-76-C-0087

FINAL REPORT

VOLUME IV

APPLICATION OF DDP TECHNOLOGY TO BMD:
ARCHITECTURES AND ALGORITHMS

For
BMDATC
Ballistic Missile Defense
Advanced Technology Center
Huntsville, Alabama 35807

Honeywell
SYSTEMS & RESEARCH CENTER
2600 RIDGWAY PARKWAY
MINNEAPOLIS, MINNESOTA 55413

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (WHEN DATA ENTERED)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOV'T ACCESSION NUMBER	3. RECIPIENT'S CATALOG NUMBER
6 4. TITLE (AND SUBTITLE) Distributed Data Processing Technology. Volume IV. Application of DDP Technology to BMD: Architectures and Algorithms.		9 5. TYPE OF REPORT/PERIOD COVERED Final Report, Oct[redacted] 1976 ~~to~~ Oct[redacted] 1977,
		14 6. PERFORMING ORG. REPORT NUMBER 77SRC68
10 7. AUTHOR(S) M.G./Gouda, Y.W./Han, E.D./Jensen, W.D./Johnson, R.Y./Kain		15 8. CONTRACT OR GRANT NUMBER(S) DASG60-76-C-0087
9. PERFORMING ORGANIZATIONS NAME/ADDRESS Honeywell Systems & Research Center 2600 Ridgway Parkway Minneapolis, Minnesota 55413		10. PROGRAM ELEMENT,PROJECT,TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME/ADDRESS Ballistic Missile Defense Advanced Technology Center Huntsville, Alabama 35807		11 12. REPORT DATE Sep[redacted] 1977
		13. NUMBER OF PAGES 191
14. MONITORING AGENCY NAME/ADDRESS (IF DIFFERENT FROM CONT. OFF.) 12 190p.		15. SECURITY CLASSIFICATION (OF THIS REPORT) Unclassified
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (OF THIS REPORT)

Approved for public release; distribution unlimited.

D D C
DEC 8 1977
E

17. DISTRIBUTION STATEMENT (OF THE ABSTRACT ENTERED IN BLOCK 20, IF DIFFERENT FROM REPORT)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (CONTINUE ON REVERSE SIDE IF NECESSARY AND IDENTIFY BY BLOCK NUMBER)

Data Processing
Computer Systems
Distributed Data Processing
Computer Architecture
Computer System Design Methodologies

20. ABSTRACT (CONTINUE ON REVERSE SIDE IF NECESSARY AND IDENTIFY BY BLOCK NUMBER)

This volume presents the results of research on techniques to develop distributed data processing architectures for specific functions, with particular reference to BMD functions. It describes two methodologies and six designs for two BMD functions. Short discussions of two nonfunctional payoffs are included to aid future workers on these problems. Many suggestions for future research and development activities are given.

HD-168 REV 11/74

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 55 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (WHEN DATA ENTERED)

402 349 LB

FOREWORD

The research documented in this volume was conducted under Ballistic Missile Defense Advanced Technology Center contract number DASG60-76-C-0087, entitled "Distributed Data Processing Technology." The work was performed by Honeywell Systems and Research Center, Minneapolis, Minnesota under the direction of Mr. C. R. Vick, Director, Data Processing Directorate, Ballistic Missile Defense Advance Technology Center. Mr. J. Scalf was the BMDATC project engineer for this contract; Ms. B. C. Stewart was the Honeywell/GRC program manager.

This report covers work from October 1976 to October 1977.

This volume presents the results of research on techniques to develop distributed data processing architectures for specific functions, with particular reference to BMD functions. It describes two methodologies and six designs for two BMD functions. Short discussions of two nonfunctional payoffs are included to aid future workers on these problems. Many suggestions for future research and development activities are given.

The work discussed in this volume was performed by M. G. Gouda, Y. W. Han, E. D. Jensen, and W. D. Johnson of the Honeywell Systems and Research Center, R. Y. Kain of the University of Minnesota, and L. Widdoes of Stanford University who acted as consultant. All of these individuals except L. Widdoes contributed to the preliminary draft of this volume; the final draft was coordinated and primarily written by R. Y. Kain.

This document is Volume IV of the final report. Other volumes of the report are the following:*

Volume I	-	Management Summary
Volume II	-	DDP Rationale: The Program Planning Point of View
Volume III	-	DDP Rationale: The Technology Point of View
Volume V	-	Application of DDP Technology to BMD: DDP Subsystem Design Requirements
Volume VI	-	Application of DDP Technology to BMD: Impact on Current DP Subsystem Design and Development Technologies
Volume VII	-	Application of DDP Technology to BMD: Experiments
Volume VIII	-	Application of DDP Technology to BMD: Research Performance Measurement
Volume IX	-	DDP Rationale: The Program Experience Point of View

---

* **Volumes V, VI, the appendix to Volume VII, and a section of Volume VIII were prepared by General Research Corporation.**

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

CONTENTS (continued)

CONTENTS (concluded)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (continued)

LIST OF ILLUSTRATIONS (continued)

LIST OF ILLUSTRATIONS (concluded)

LIST OF TABLES

SECTION 1

INTRODUCTION

1.1 OVERVIEW

Research concerning the direct mapping of BMD functions onto architectures is discussed in this volume. Key results of the research are summarized in Section 6. A large number of research issues and recommendations are presented in Section 7. The remainder of the volume is devoted to presentation of detailed "design theory" research results at the node and processor levels. Section 2 contains two preliminary formal methodologies for BMD function/architecture pairing; Sections 3 and 4 evaluate conventional architectures and propose novel architectures for two BMD functions (radar scheduling and optical discrimination); Section 5 discusses results of the integration of two payoffs (graceful degradation and graceful growth) into the architecture/function pairing methodology.

1.2 OBJECTIVE

The architectural development task was undertaken in order to:

1. Understand the design process used by system designers. This knowledge would advance later efforts to improve design aids and specification techniques.
2. Document design methodologies to direct designers as they develop system architectures. An understanding of the methods would also suggest further efforts in this area.

3. Ascertain benefits of node- and processor-level distributed data processing for BMD. A preliminary indication would provide direction for further efforts.

4. Uncover descriptive needs. Known descriptive techniques could be adapted once the difficulties of their use in DDP design situations had been determined.

5. Understand the design interfaces. Knowledge of how designers relate to others in the specification/design/implementation sequence assists in clarifying descriptive needs.

6. Suggest experiments to improve our knowledge about design options and payoffs of DDP for BMD.

7. Suggest design techniques that enhance the distributed data processing (DDP) payoffs for Ballistic Missile Defense (BMD) systems. A catalog of documented design techniques would make the designer's task more straightforward.

1.3 APPROACH

During this project, two BMD functions were selected and presented as design problems to several designers of diverse backgrounds and experience; two of the designers documented their design methodologies and one suggested techniques to improve reliability, etc.[1] An introspective review of these efforts uncovered needs and requirements for further research, development, and experimentation in node and processor architecture design.

SECTION 2

METHODOLOGIES

The art of systems design will continue as an art in the forseeable future. Methodologies will be developed to direct the designer, and automated aids will assist him in the design process; however, the heuristic (artistic) steps will remain.

In the following sections, we discuss two different methodologies developed during this effort. These methodologies have been designed to guide designers in taking a specification of a real-time function and producing a distributed data processing architecture which not only performs the desired function, but also meets nonfunctional requirements such as cost, reliability, and graceful growth. Neither methodology should be considered a precise algorithm but must be viewed as a set of directions to aid the designer.

Architectures developed for the radar scheduling function using these methodologies are discussed in Sections 3.3.3 and 3.3.4.

2.1 DESIGNER C's METHODOLOGY

Designer C's * methodology relies on the notions of function decomposition, architecture graphs, and elements composition. These are discussed in

*The five designers who worked on architectural development are identified as "designer A" through "designer E," with the letters roughly indicating the chronological order of their efforts.

the following subsections. With these preliminaries out of the way, we present the methodology in Section 2.1.4 and an example of its application in Section 2.1.5. This methodology was applied to the radar scheduling BMD function, as discussed in Section 3.3.3.

2.1.1 Function Decomposition

Informally, a function specifies a computation applied to a data structure; this relationship is depicted in Figure 1a, where F operates on data structure D. Some functions may be appropriate for implementation in a single processing element of a given technology, but, when that is not the case (due, for example, to processing rate requirements), the function must be decomposed before it can be implemented using the particular technology. Thus we consider the decomposition of a function into a collection of smaller functions called function components, which together implement the original function operating on the original data structure.

Coordination among the function components is achieved by control messages passed among the components, as illustrated in Figure 1b.

To improve protection, performance, and modularity, it may be desirable to construct the component functions so that each operates on its own local data structure. Many of these structures will not be shared among function components, but some structures may be replicated. Data messages may have to be exchanged between component functions to achieve the structural decomposition. Figure 1c shows a decomposition of the function F into three components, each operating on a local data structure.

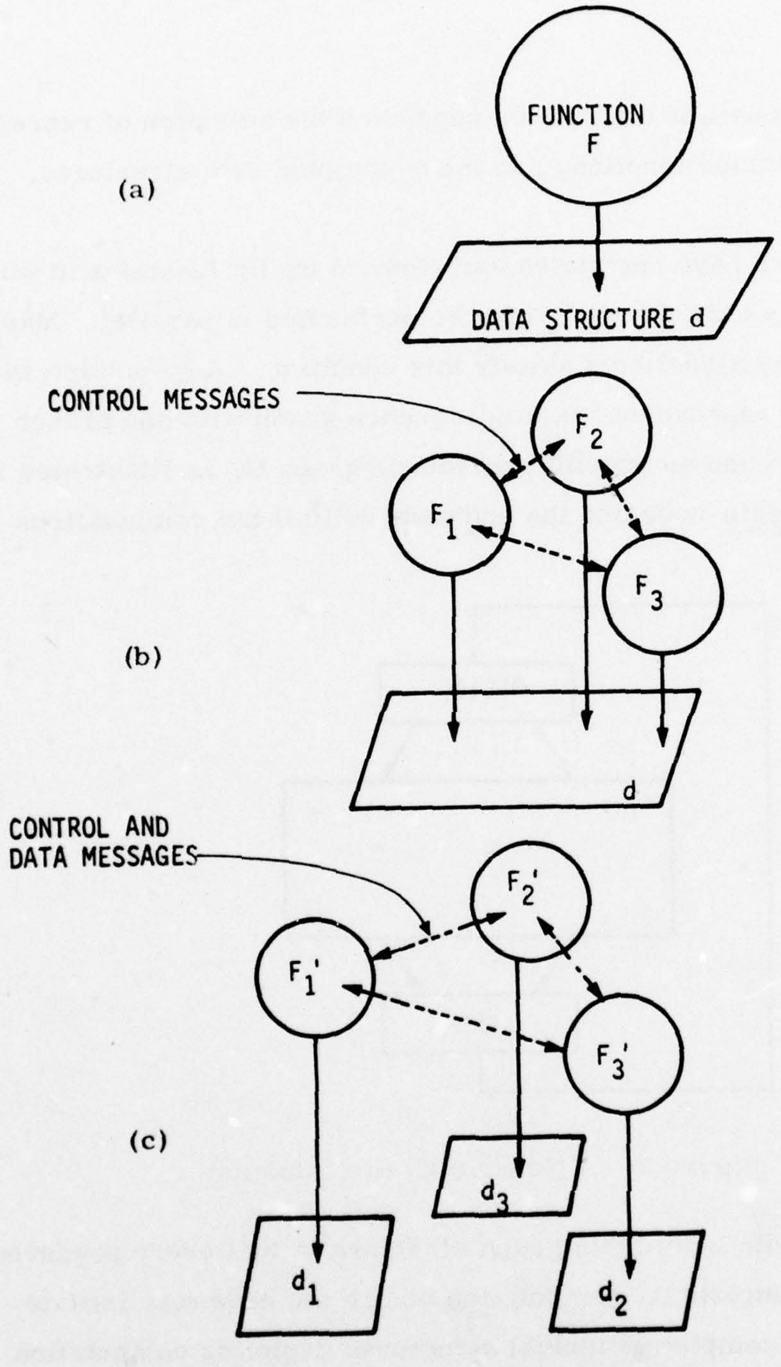


Figure 1. Function and Data Decomposition

Details of decomposition algorithms depend on the selection of representation techniques to describe functions and the associated data structures.

For this study, we have restricted our study to cyclic functions in which some operations within the cycle may be performed in parallel. Many real-time systems algorithms satisfy this condition. Any function within this class can be represented as a precedence graph with one branch forming a loop around an acyclic precedence graph G, as illustrated in Figure 2. The begin node and the end node delimit the computations within the loop.

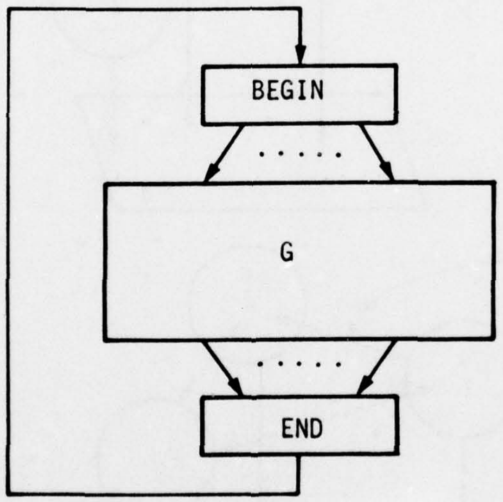


Figure 2. A Generic Cyclic Function

The precedence rule within this graph structure is that every predecessor of a node must complete its computation before the node may initiate execution. More complex graphical structures depicting computation structures have been studied by Dennis and others.[2-10] Here we restrict ourselves to the simple case which is sufficient for interesting BMD functions.

Figure 3 illustrates a specific function constructed from five intermediate subfunctions. The optimum scheduling of the execution of the subfunctions on a given set of processors is a difficult problem; in fact most cases are NP-complete[11-16] which means that non-optimum schedules will have to be tolerated. Regardless of the scheduling policy or algorithm, the schedule must meet the precedence constraints expressed in the function graph. Figure 4 shows one feasible schedule for the execution of the function of Figure 3 on two processors, assuming that every node requires the same execution time. We assume the begin and end nodes represent synchronization operations that require no time.

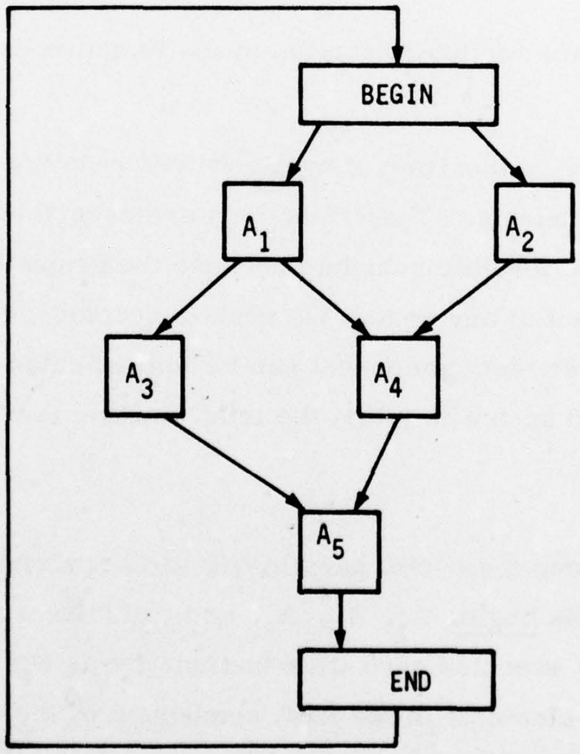


Figure 3. A Specific Cyclic Function

The arc from end to begin forces every feasible schedule to consist of cycles; in each cycle, each node within the precedence graph is executed once. Thus we call these functions cyclic.

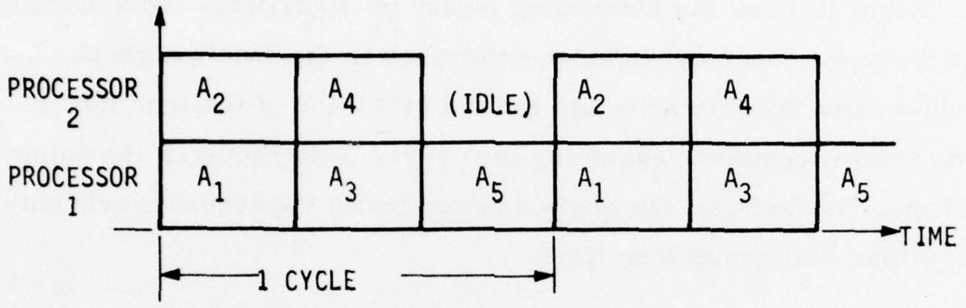


Figure 4. A Schedule for the Execution of the Function of Figure 3

Now we consider the decomposition of cyclic functions into a set of sequential cyclic components. These functions are described by graphs that allow only a single feasible schedule because the graph does not exhibit fan-in or fan-out at any node. We wish to decompose the cyclic parallel function so that each component can be implemented on a separate sequential machine, in such a way that the total function is realized by the distributed system.

As an example, consider the cyclic parallel function described by the graph in Figure 3. The nodes begin, A_1, A_3, A_5, and end form a sequential directed path which is executed each time the function is executed. We select these nodes as elements of the first component of the graph and mark the links connecting them (in Figure 5) with thick lines to indicate that they belong to one component of the decomposition. The remaining nodes (A_2 and A_4) belong to the second component.

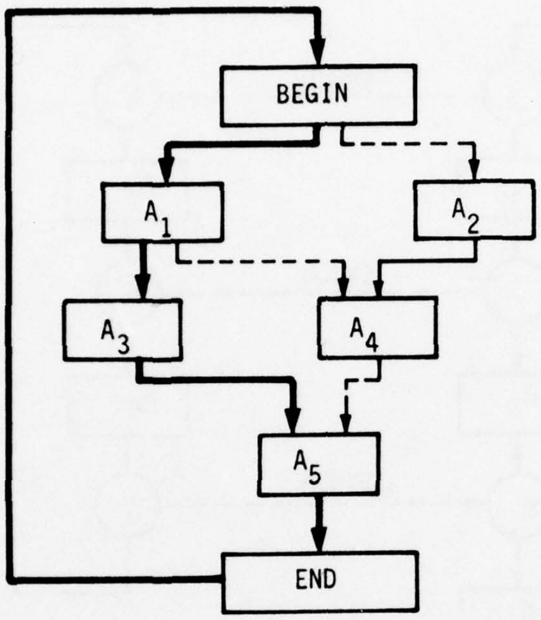


Figure 5. One Component Selected from a Cyclic Parallel Function

Figure 6 exhibits both components of the decomposition and shows the necessary synchronization operations. A circle containing "S" indicates sending a synchronization message along a dashed link to a receiver "R" which must await the synchronization message before it may activate its successor. It is straightforward to verify that the communication between the two components is bounded and deadlock-free[17] since the communications can be represented as a safe and live marked directed graph.[8, 18, 19]

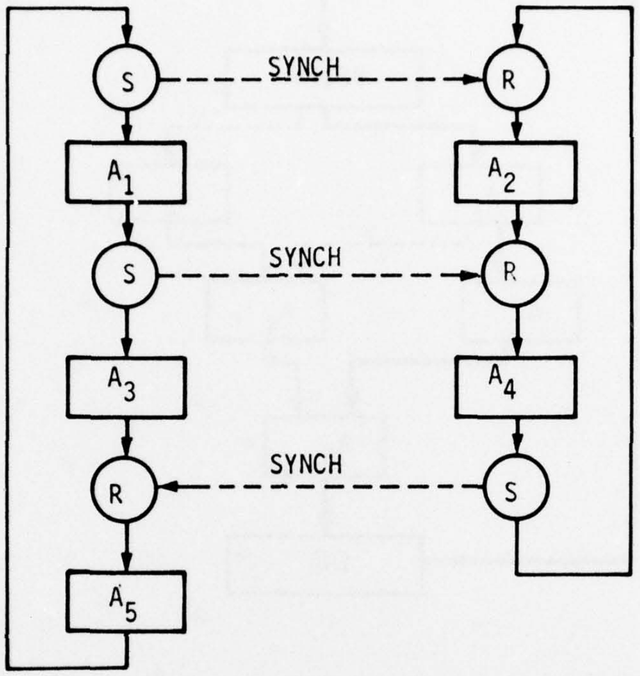


Figure 6. A Complete Decomposition of the Function from Figure 3, Including Synchronization

If the number of machine instructions needed for each operation can be estimated, and if an execution rate requirement is imposed, we can determine both the processing rates P_i (instructions/second) for each component function and the communication rates c_{ij} (bits/second) between each pair of component functions. Schematically we represent the configuration of two communicating processors as shown in Figure 7.

In the next section, we expand the interconnection and computational structures to form architecture graphs (Figure 7 is an example of a preliminary stage in the development of an architecture graph).

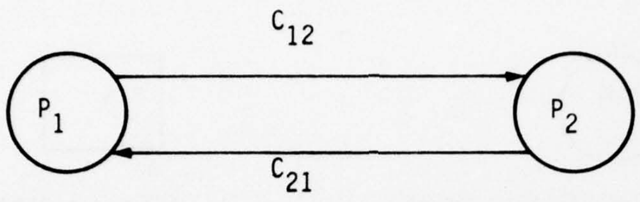
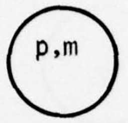


Figure 7. Two Communicating Processors

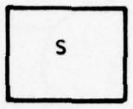
2.1.2 Architecture Graphs

Architecture graphs are useful tools to specify the processing, storage, and communication requirements of functions. An architecture graph is a directed graph whose nodes represent the system components and whose links represent the "connections" between the different components. The functional requirements of system components and connections label the corresponding nodes and edges in the architecture graph.

There are two types of nodes in an architecture graph: processing units and memory units. A processing unit (Figure 8a) is represented as a circle labeled with an ordered pair ($\underline{p}$, $\underline{m}$) where $\underline{p}$ is the processing rate (in instructions/second) of the processing unit, and $\underline{m}$ is its memory size (in bits or words). A memory unit (Figure 8b) is represented as a rectangle, labeled with its memory size s.



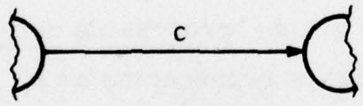
(a) PROCESSING UNIT



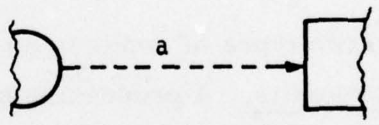
(b) MEMORY UNIT

Figure 8. Two Types of Nodes in Architecture Graphs

There are two types of edges in an architecture graph: communication links and access links. A communication link (Figure 9a) is represented as a solid line; it connects two processing units and is labeled with the communication rate $\underline{c}$ in bits/second. A communication link has one or two directions to show in which directions messages may flow along the link.



(a) COMMUNICATION LINK



(b) ACCESS LINK

Figure 9. Two Types of Links in Architecture Graphs

An access link (Figure 9b) is represented as a dashed line; it connects one processing unit to one memory unit and is labeled with the access rate a (in bits/second). The direction(s) of an access link indicates the access mode(s) (read or write).

Figure 10 shows an architecture graph with five processing elements (labeled with pairs $(p_1, m_1) \ldots (p_5, m_5)$) and four memory elements (labeled with s_1, s_2, s_3, and s_4). The graph has seven communication links which include one input link i_1 and two output links o_1 and o_2. Also, the graph has seven access links.

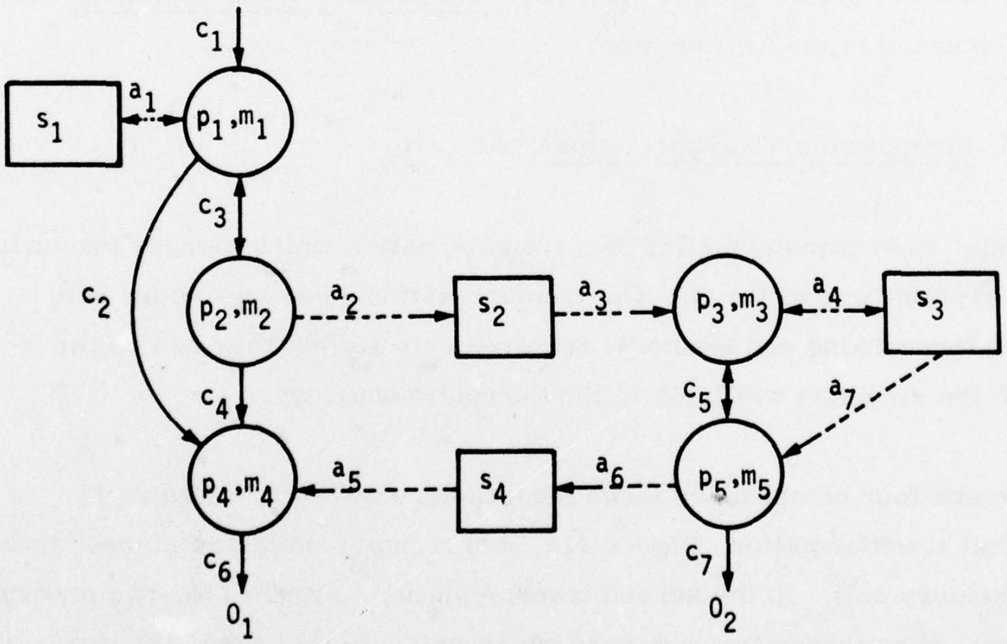


Figure 10. An Architecture Graph

Let G be an architecture graph. To realize G as a distributed computer system, its nodes must be realized using processing and memory elements available from available technology, such that the specified processing and memory requirements are met. These elements are connected (as indicated in G), such that the communication and accessing requirements are met using available technology.

However, if the nodes in G have small processing and memory requirements, then, to realize G more efficiently, the designer should consider merging its nodes together into nodes with larger requirements which can be realized economically by available elements. The transformation merging nodes in architecture graphs are called composition transformations and are discussed in the next section.

2.1.3 Composition Transformations

A composition transformation is a transformation which merges two nodes of an architecture graph G. The transformation generates nodes with larger (processing and memory) requirements so that they can better match the elements available in the current technology.

There are four composition transformations, as shown in Figure 11. In the first transformation, Figure 11a, two memory units are merged into one memory unit. In the second transformation, Figure 11b, two processing units are merged into one processing unit. In this case, the two original processing units may be implemented as virtual units hosted within the large processing unit. This implementation incurs an overhead which is reflected in increased processing and memory requirements, i.e.,

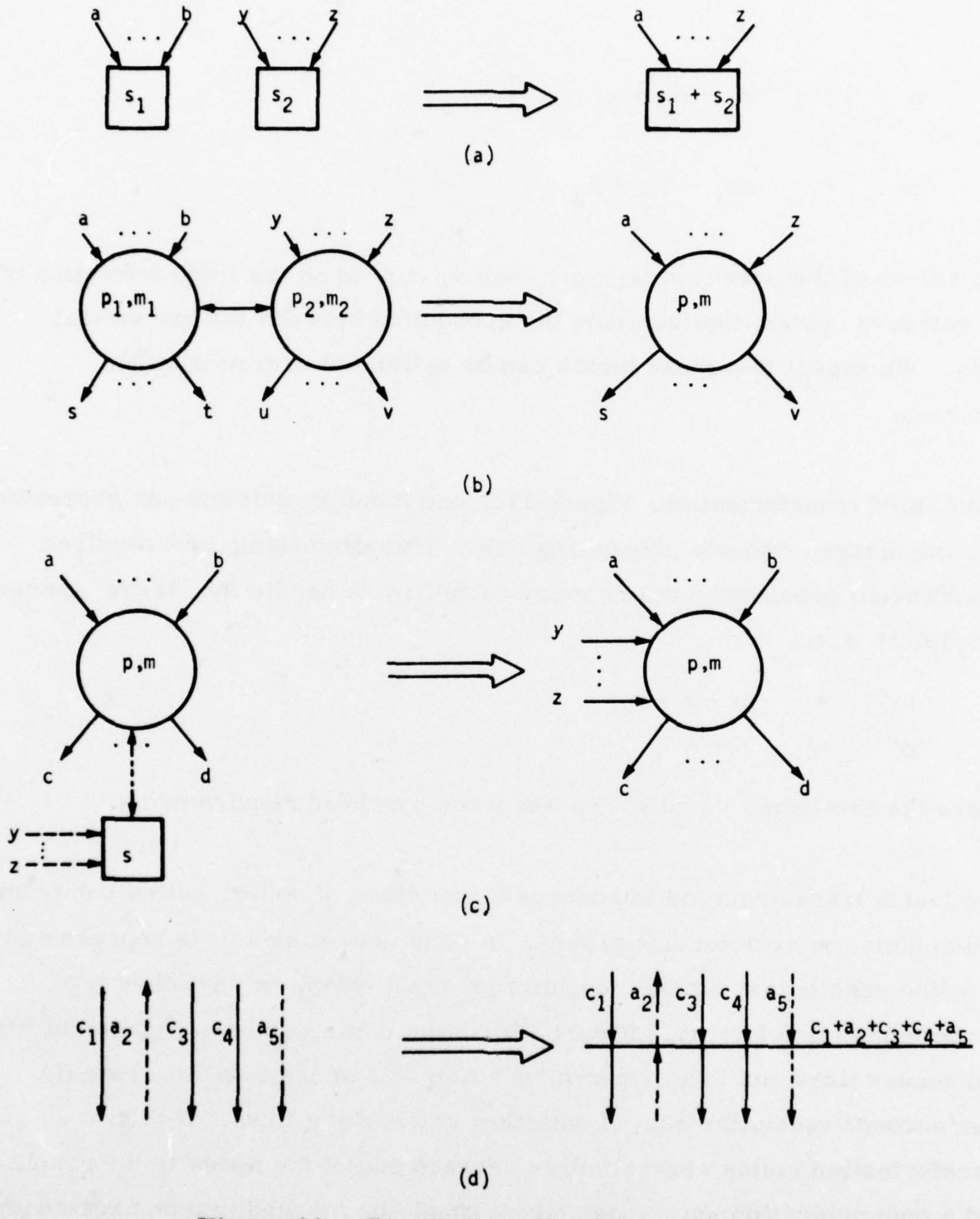


Figure 11. Composition Transformations

$$p = p_1 + p_2 + v_1$$

$$m = m_1 + m_2 + v_2$$

The values of the overhead terms v_1 and v_2 depend on the implementation of the software system that controls the scheduling between the two virtual units. We expect that these terms can be estimated with reasonable accuracy.

In the third transformation, Figure 11c, one memory unit and one processing unit are merged into one processing unit. This processing unit requires an additional processing and memory capability to handle the "large" memory added to it, i.e.,

$$m' = m + s + v$$
$$p' = p + v'$$

where the two terms v and v' represent the overhead requirements.

The fourth transformation introduces a new class of nodes, called communication units, to architecture graphs. A communication unit is represented as a line segment; it can be thought of as a bus, loop, or any other type of communication facility. Figure 11d shows how a number of communication and access links can form a communication unit whose data flow rate (in bits/second) equals the sum of data flow rates of the links. If this transformation yields several edges between one of the nodes in the graph and a communication unit, those edges should be merged into one edge with data flow rate equal to the sum of the rates associated with the edges. This transformation is applicable only when it does not introduce an edge from one communication unit to another communication unit.

A sequence of these composition transformations applied to any architecture graph produces a reduced version of the graph. Obviously, if we begin with the same graph G and apply different sequences of composition transformations, we get a number of different architecture graphs; all of them are reduced versions of G. This observation is used in the Designer C's methodology.

Composition transformations can be viewed as the reverse of function decomposition, as defined earlier, with one basic difference. Function decomposition is a logical process in which a function is decomposed into its "basic" components, while composition is an optimization process to reach the best packing for the function into the components of a distributed system. Both function decomposition and composition transformations are used in defining the following methodology for DDP system design.

2.1.4 A Design Methodology for Distributed Systems

Figure 12 outlines a design methodology of distributed computer systems. It begins with a function description and ends with a reliable architecture which satisfies its function requirements and has a number of "good" characteristics. The methodology is as follows:

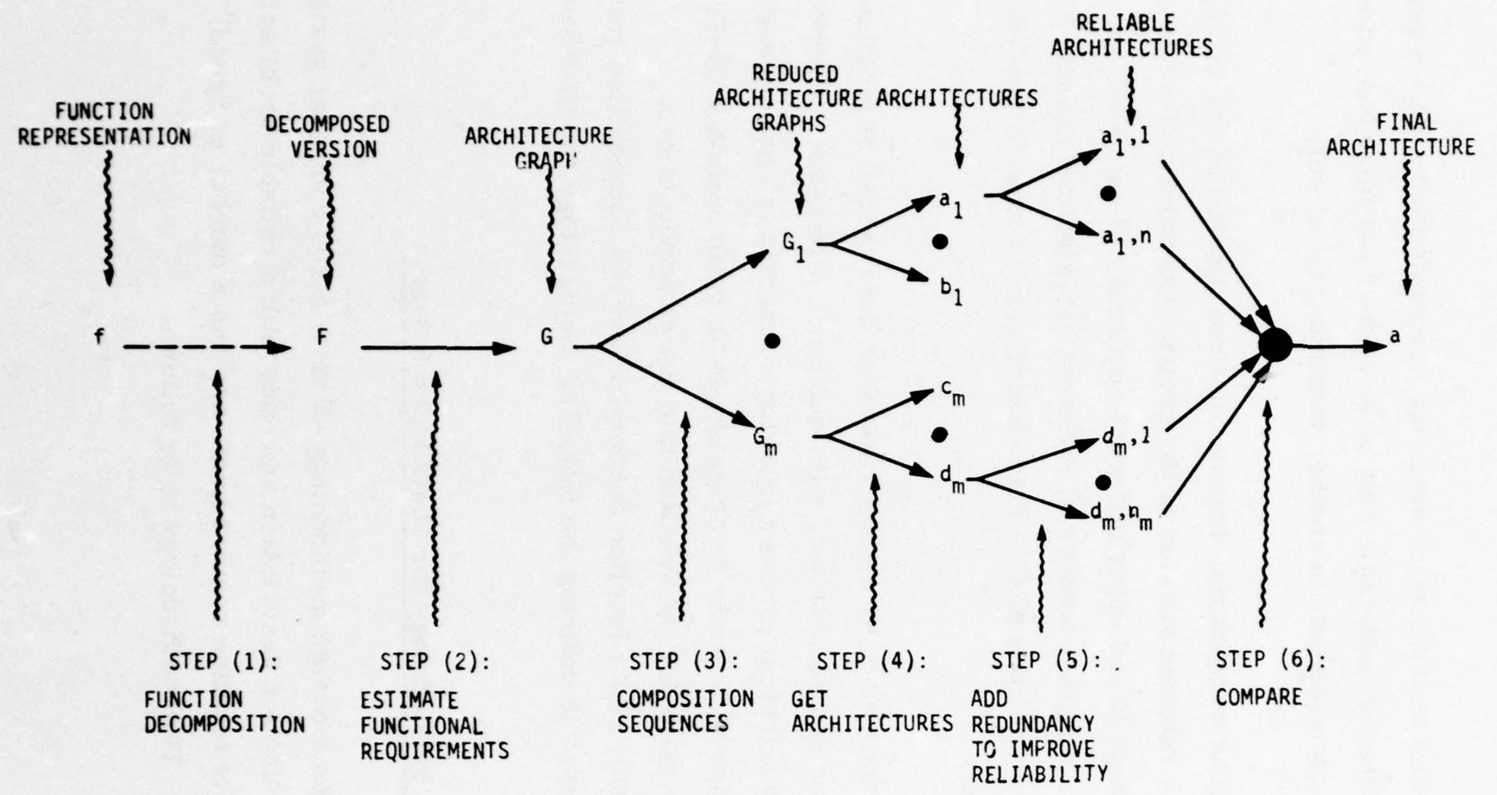

FUNCTION REPRESENTATION
DECOMPOSED VERSION
ARCHITECTURE GRAPH
REDUCED ARCHITECTURE GRAPHS
ARCHITECTURES
RELIABLE ARCHITECTURES
FINAL ARCHITECTURE
f
F
G
G_1
G_m
a_1
b_1
c_m
d_m
$a_{1,1}$
$a_{1,n}$
$d_{m,1}$
d_{m,n_m}
a
STEP (1): FUNCTION DECOMPOSITION
STEP (2): ESTIMATE FUNCTIONAL REQUIREMENTS
STEP (3): COMPOSITION SEQUENCES
STEP (4): GET ARCHITECTURES
STEP (5): ADD REDUNDANCY TO IMPROVE RELIABILITY
STEP (6): COMPARE

Figure 12. An Outline of the Design Methodology

Step 1--Decompose the function into a number of small interacting components. Let F be the representation of the function after the decomposition.

Step 2--Represent each component in F as a node in an architecture graph G. The label of each node reflects an estimate of the functional requirements of the corresponding component. Edges are added between the nodes of G to signify the interactions between the corresponding components, and their labels are estimates of the interaction rates.

Step 3--Apply a number of composition sequences to G to generate reduced architecture graphs G_1, G_2, ..., G_m.

Step 4--For each reduced architecture graph G, get a number of corresponding architectures a_i as follows. The nodes of G_i are mapped into processing and memory units in a_i. Communication units and links in G_i are mapped into communication paths such as dedicated buses, shared buses or loops in a_i.

Step 5--Add redundancy to each architecture a_i to improve its reliability. Since redundancy can be added in different ways, the designer will create a number of reliable distributed architecutres $a_{i,1}$, ..., a_{i,n_i} for each architecture a_i.

Step 6--The resulting architectures are compared with respect to their costs, their reliabilities, and other important design criteria. A final architecture is chosen.

Next, we discuss briefly how this methodology is applied to a small example.

2.1.5 An Example

Figure 13 shows an example where this design methodology is applied to a simple function. The function is shown in Figure 13a. It is a cyclic and parallel function as defined in Section 2.1.3, with the exception that one of its operations, B_1, uses a file to store some parameter histories.

The function can be decomposed into two interacting components as shown in Figure 13b. In this case, the architecture graph can be constructed as in Figure 13c. In the architecture graphs, each processing unit corresponds to one function component, and the memory unit corresponds to the system file. A number of composition transformations can be applied on the architecture graph to get the reduced graphs shown in Figure 13d.

Now, every node of each reduced graph must be realized using an available element. Then the designer estimates the total payoffs realized by each reduced graph and chooses the reduced graph which yields the maximum benefit.

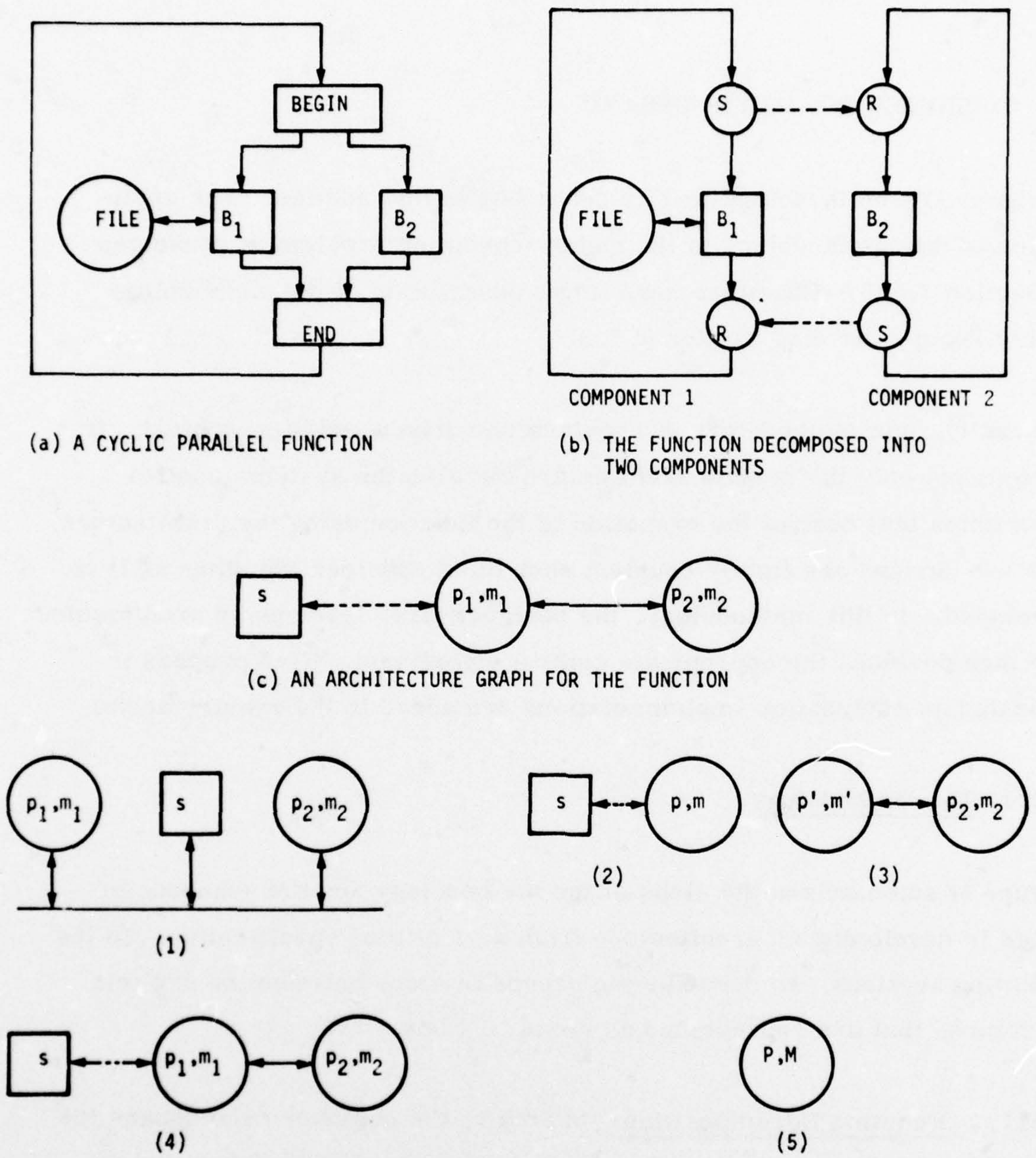

BEGIN
FILE
B_1
B_2
END
(a) A CYCLIC PARALLEL FUNCTION
S
R
FILE
B_1
B_2
R
S
COMPONENT 1
COMPONENT 2
(b) THE FUNCTION DECOMPOSED INTO TWO COMPONENTS
s
p_1,m_1
p_2,m_2
(c) AN ARCHITECTURE GRAPH FOR THE FUNCTION
p_1,m_1
s
p_2,m_2
(1)
s
p,m
(2)
p',m'
p_2,m_2
(3)
s
p_1,m_1
p_2,m_2
(4)
P,M
(5)
(d) A NUMBER OF REDUCED ARCHITECTURE GRAPHS

Figure 13. An Example

2.2 DESIGNER D's METHODOLOGY

Designer D's methodology will be described in this section. The application of this methodology to the radar scheduling problem is presented in Section 3.3.3. Examples used in the description of the methodology do not require reading Section 3.3.3.

Generally, this methodology is requirement-driven and hierarchical. It designs not only the module architecture but also the systems control algorithms that control the execution of the function using the architecture. The two designs are tightly coupled; each must consider the other as it is developed. In this methodology, the designer first develops an architecture and then develops the appropriate control algorithms. This process is repeated as subfunction implementations are added to the requirements.

2.2.1 The Methodology

Figure 14 summarizes the steps of the methodology and the sequence of usage in developing an architecture from a functional specification. In the following sections, we describe the groups of steps between the explicit structures that are represented as nodes in Figure 14.

2.2.1.1 Function Decomposition--In Step 1, the designer decomposes the given function into a set of subfunctions; he studies the given function to select useful subfunctions. He should include all significant computations in subfunctions, except high-level control functions. Without module

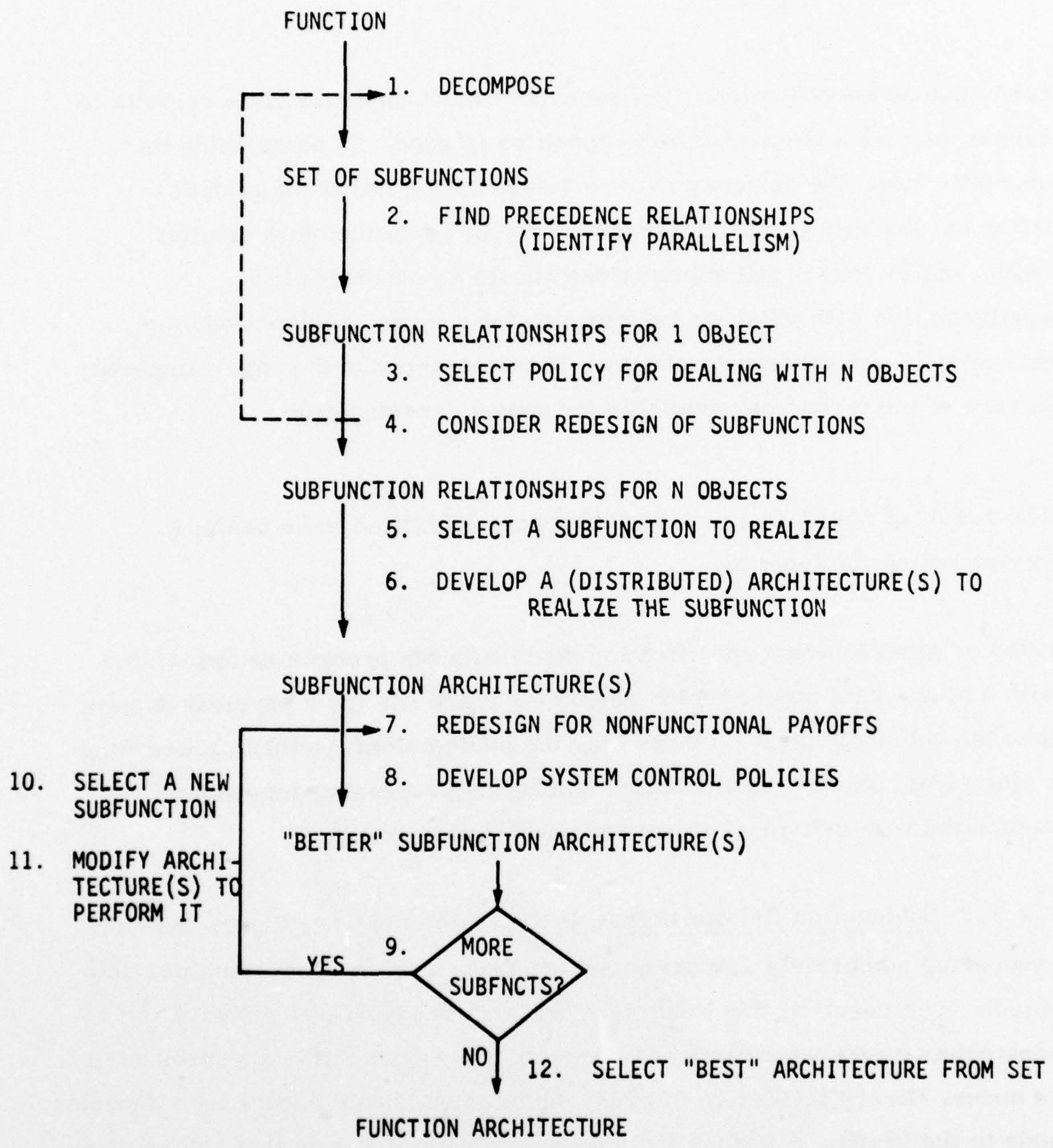

FUNCTION
1. DECOMPOSE
SET OF SUBFUNCTIONS
2. FIND PRECEDENCE RELATIONSHIPS (IDENTIFY PARALLELISM)
SUBFUNCTION RELATIONSHIPS FOR 1 OBJECT
3. SELECT POLICY FOR DEALING WITH N OBJECTS
4. CONSIDER REDESIGN OF SUBFUNCTIONS
SUBFUNCTION RELATIONSHIPS FOR N OBJECTS
5. SELECT A SUBFUNCTION TO REALIZE
6. DEVELOP A (DISTRIBUTED) ARCHITECTURE(S) TO REALIZE THE SUBFUNCTION
SUBFUNCTION ARCHITECTURE(S)
7. REDESIGN FOR NONFUNCTIONAL PAYOFFS
8. DEVELOP SYSTEM CONTROL POLICIES
10. SELECT A NEW SUBFUNCTION
11. MODIFY ARCHITECTURE(S) TO PERFORM IT
"BETTER" SUBFUNCTION ARCHITECTURE(S)
9. MORE SUBFNCTS?
YES
NO
12. SELECT "BEST" ARCHITECTURE FROM SET
FUNCTION ARCHITECTURE

Figure 14. Designer D's Methodology

performance specifications, the designer cannot apply absolute criteria to decide whether a particular decomposition is good. In performing this heuristic step, the designer may be guided by the following guidelines: 1) any excessively complex subfunction might be divided into smaller units, and 2) very small subfunctions should be considered for amalgamation with other subfunctions to form larger subfunctions (as this reduces the system overhead). The designer should gauge complexity in view of the technology available for module construction.

The result of Step 1 is a set of subfunctions identified to be useful in implementing the function.

If the original function specification deals with the processing associated with a single data object (which could be a track file for a physical object), then the subfunction set includes only the subfunctions useful in processing a single data object. Coordination among the processing sequences associated with individual objects is developed in Step 3.

2.2.1.2 Subfunction Relationships--In Step 2 the designer determines any sequencing constraints among the subfunctions used in processing one data object. The result of this study is a precedence graph that displays the constraints as directed branches connecting the subfunctions, represented as nodes. Every invocation of every subfunction should appear as a separate node in the graph. A branch from node A to node B means that subfunction B cannot be performed until subfunction A has been completed. In the simplest cases, all processor subfunctions must complete processing before the successor subfunction can initiate processing. More elaborate graphical representations can be used (see References 2-10) but were not explicitly addressed in this methodology.

If the designer produces a graph showing the minimum necessary sequencing constraints, then he has also identified all possible parallelism among the subfunctions because any two subfunctions not connected by a path in the graph can be performed in parallel. For example, Figure 15 illustrates a precedence graph in which subfunctions are denoted by capital letters and in which the following minimum precedence constraints are represented:

A precedes B

B precedes D

C precedes E

D precedes E

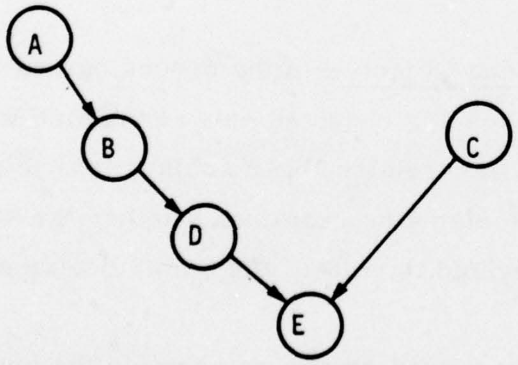


Figure 15. A Precedence Graph

Precedence constraints can be represented as inequalities ($A < B$ means that A must precede B). In this notation, the designer can conveniently derive other precedence constraints; for example, from $A < B$ and $B < D$ one can conclude $A < D$ (i.e., precedence is transitive).

Precedence relationships may not enforce a total ordering on the nodes in the graph. In a total ordering, either $A < B$ or $B < A$ for any pair of nodes (A, B). Note that a total ordering of subfunction processing requires sequential execution of the subfunctions (this does not require sequential processing in the complete system since the implementations of the subfunctions may use parallel processing). If neither $A < B$ nor $B < A$, then the functions A and B may be performed in parallel.

In this step the designer identifies all possible parallelism, but he does not select a scheduling strategy until Step 8, after he has considered how to deal with N objects (Step 3) and has developed some subfunction architectures (Step 7).

2.2.1.3 Find Relationships Among Objects--In the preceding two steps, the designer has studied the processing requirements associated with a single data object. In Step 3 he interrelates these subfunctions to process a complete set of objects, and in Step 4 he examines whether the subfunction breakdown should be revised in view of the Step 3 decisions.

Generally, in Step 3 the designer considers how to combine the processing requirements for a set of data objects into a coherent efficient scheme for handling a set of similar objects. For example, knowing how to discriminate the object described in a single track file, he now concludes how best to handle a set of objects described by a set of track files. If the function does not require a multiplicity of objects, the designer skips Step 3. Possible Step 3 strategies include the following:

1. Include a processing element for each data object and process all objects in parallel.
2. Design a pipelined machine and place all data objects in a vector for processing sequentially but quickly.
3. Include a set of processing elements and assign a subset of the set of objects to each processor.

More complex strategies are possible when the subfunctions use unequal amounts of processing capability. An example of this strategy is shown in Section 3. 3. 4.

Decisions about processing a set are very important when the system requirements include the possibility of growth in the number of data objects that must be processed. Growth questions are discussed further in Section 5. 2.

In Step 3 the designer may feel that he does not have sufficient information to select among the options; in this case, he chooses several attractive selections and proceeds with several optional designs.

In Step 4 the designer considers whether any of the options look reasonable. If none seem to be so, he starts anew at Step 1. This decision is very imprecise; the designer is expected to rely on his intuition and experience.

The designer should constantly be aware that his design may be unrealistic; at every step he should consider whether to modify previous design decisions. This step is explicitly shown since the designer may not realize the import of his functional decomposition decisions until he attemps to modify the design for a multiplicity of objects.

Now the designer has a complete decomposition of the problem into a set of subfunctions; in the remaining steps he will construct hardware and software architectures to implement the function.

2.2.1.4 Develop a Subfunction Architecture--In the next two steps the designer specifies an architecture that realizes one subfunction. Steps 7 through 11 form a cycle in which an additional subfunction is included in the design upon every pass through the cycle. This cyclic process is described in Section 2.2.1.5.

The designer must select, in Step 5, which subfunction shall be realized first. Intuition and experience are required for this selection; the methodology offers no help with this problem.

Once the subfunction is selected, the designer chooses, in Step 6, a set of architectures that are good candidates for realizing the subfunction effectively and efficiently. In this step, the designer should consider the next level of detail of the subfunction. He may, for example, discover that the subfunction itself consists of six potentially parallel sequences which feed a seventh sequence. A seven-processor architecture as shown in Figure 16 might be appropriate.

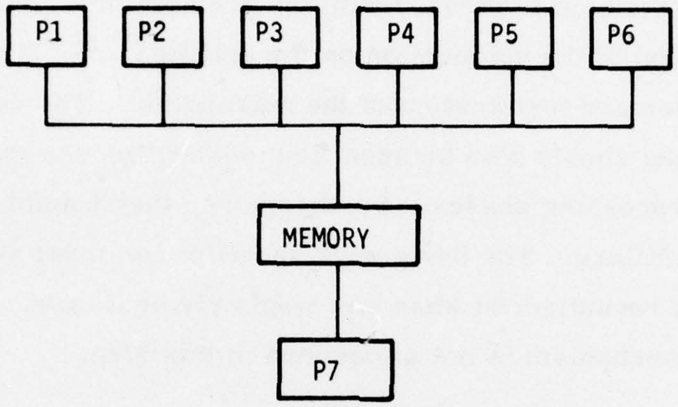


Figure 16. A Representative Subfunction Architecture

2.2.1.5 <u>Improve Architectures and Incorporate Other Subfunctions</u>--Steps 7 through 11 form a cycle in which the designer improves a given architecture and then includes another subfunction in the design. He must loop through this cycle until all subfunctions have been realized.

In Step 7 the designer must determine the properties of the design with respect to reliability, flexibility (ease of expansion, graceful degradation, graceful saturation), and other payoffs not related to the functionality of the system. These properties depend upon the corresponding properties of the component modules; some of the relationships among these properties are discussed in Volume 2 of this report.

In Step 8 the designer develops preliminary specifications concerning system control algorithms to be used with the architecture. These algorithms control the execution of the subfunction on the architecture. Also, they control the initiation and termination of the subfunction. The use of communication links should also be specified; several modes may be used during different processing phases. Furthermore, they should perform during component failure. The designer's specification must state how the system will be reconfigured when any single element fails. The failure detection mechanism is not of concern in this step.

Step 9 is the trivial decision whether there are remaining subfunctions that have not been incorporated into the design. If all subfunctions have been incorporated, the design is complete; the next step is Step 12. Otherwise, the designer proceeds to Step 10.

The designer, in Step 10, selects another subfunction to be incorporated into the design. The methodology does not suggest how he can best select the appropriate subfunction. Again, his experience and intuition must be his guide.

Once the new subfunction is selected, the architecture is redesigned in Step 11 to add the new subfunction to the set already implemented. The designer must consider system control algorithms and overall system timing. If, for example, there exist time intervals when certain modules are not busy, they might be used to assist in computing the new subfunction. If the elements of the system are all allocated at all times, or if the new subfunction is not compatible with the previous architecture, the designer has the option to create a new element structure with little relationship to the existing system. The fact that this type of decision is wise should not be interpreted as a negative observation concerning the previous design; the function may simply be constructed from several internally compatible sets of subfunctions even though those sets are not compatible with each other.

After incorporating the added subfunction, the designer returns to Step 7 where he reconsiders the nonfunctional payoffs. One would expect that succeeding passes through Step 7 would cause modifications only to the new elements in the architecture since the previous passes should have sufficiently improved the properties of the old arrangements of elements.

2.2.1.6 Final Selection--Recall that the designer may, in previous steps, have been unable to select a single structure to use in continuing the design process. Without such a decision, we recommended that he perform the remaining steps for all attractive options until the selection became obvious. If several options remain after the designs are all completed, the designer must use "goodness" measures to evaluate all competitors; this process should select the final architecture from the candidates.

This twelfth step completes the design process.

2.2.2 Observations

With hindsight we observe that this methodology, while complete, emphasizes processing requirements. Memories are considered as an adjunct of each processing unit.

As with all methodologies, this provides structure for the design process but does not tell the designer how to make any difficult decisions. It places those decisions before him in a systematic manner, however, so that he cannot inadvertently skip any of them.

2.3 COMPARISONS

Both methodologies are similar in that they decompose the function into subfunctions and then realize those subfunctions. They differ, however, in that Designer D's methodology implements subfunctions using many architectural modules, while Designer C's methodology assigns subfunctions to single modules within the architecture.

Note that D's methodology does not guide the designer in implementing the subfunctions. Thus an attractive combination is one in which Designer C's methodology is used to subdivide the subfunction and then to produce a distributed realization, while Designer D's methodology provides the overall subfunction structure to implement the function.

Computer system design will continue as an art for the forseeable future. Design methodologies can be very helpful in guiding the artist through his task but will not replace his judgment based on intuition and experience. Automated aids can be used to evaluate designs and direct the designer through the appropriate sequence.

Section 3 describes a design exercise in which Designers C and D use their methodologies to develop systems architectures.

SECTION 3

RADAR SCHEDULING

Two BMD functions were used as test cases to study the design of distributed architectures. Several designers created architectures to perform these functions. These cases suggested improvements in design methodologies, improvements in specification techniques, and experiments that would improve the designer's capabilities. The two sample functions were radar scheduling, described in this section, and optical discrimination, described in Section 4.

Initially, we hoped that we would be able to develop generic architectures to perform resource scheduling and allocation of arbitrary system resources. This required that we present the designers with a top-level description of the scheduling function, without specifying a detailed algorithm. The specification is presented and discussed in the following subsection.

To show the benefits of distributed data processing for BMD, we investigated nondistributed architectures applied to the scheduling problem. Their advantages and disadvantages could be compared with the attributes of the newly-developed distributed architectures. Section 3.2 describes how several nondistributed architectures might perform the scheduling function.

Section 3.3. details the efforts of the four designers who developed architectures for radar scheduling.

The various architectures are contrasted and compared in Section 3.4. Also, lessons learned from the experience and suggestions for further studies are described. The designers identified certain experiments that might aid further architectural design efforts; these are discussed in Volume VII of this report.

3.1 THE SCHEDULING PROBLEM

All scheduling problems are equivalent to an optimization problem in which a limited number of resources are to be assigned to a number of requests. Additional constraints prohibit arbitrary scheduling. In the radar scheduling problem, these constraints arise from clutter (reflections of recent transmissions from nearby objects), power limitations in the electronics, resonances in the radar system, and so forth. The schedule should not only meet the demands and constraints but also should maximize a benefit function; this function reflects the fact that some requests are more urgent than others. The general scheduling problem is hard to solve; even with a fixed set of requests, a number of possible "good" schedules will have to be constructed and evaluated to find the best one. Note, for example, that most scheduling problems are NP-complete (see References 11-16) which means that designers cannot expect to implement optimum scheduling policies.

Radar scheduling is more complex than the simple scheduling problem because each request actually requires two intervals of radar usage: one for transmission and a later one for reception. The complexity also increases because requests arrive at unpredictable moments; the newly arriving request may require discarding any future schedule to include the new (high priority) request.

We had difficulties in determining the essence of the scheduling problem due to the "solution as requirement" syndrome aptly described by Ross and Schoman:[20]

> One of the current difficulties in requirements definition, remember, is that system developers are often charged with documenting requirements. Their design background leads then (however well-intentioned they may be) to think of system architecture rather than functional architecture, and to define requirements in terms of solutions.

It seemed that, whenever we asked what the radar scheduling problem was, we were confronted with yet another algorithm to solve the problem. These algorithms included the interlacing algorithm, the nesting algorithm, and variations of them adapted for multi-receiver radars.[21-23]

We experienced a communications problem in conveying the true requirements of this problem to the designers who developed architectures to solve it. The communications problem was not apparent until all designs were completed. Thus we did *not* rework the problem, but rather were very conscientious as we defined the discrimination problem (see Section 4.1). As a consequence of the communication problem, the designers all solved slightly different problems. Here we summarize the various versions; some versions are discussed in slightly more detail when the corresponding designs are covered in succeeding sections.

3.1.1 Algorithm Variants

There were two major types of differences among the designers. First was the specification of the classes of algorithms that should execute efficiently on the new architecture. Two types of algorithm differences concern the use of priorities and frames. Second, designers differed widely in their assumptions regarding the probabilities of certain events. This may seem a minor point, but drastic differences suggested architectures that would not be efficient if the assumptions were not correct. We discuss this issue in Section 3.4.

The designers adopted extreme positions on two algorithm variations and one probability assumption. Table 1 and Figure 17 illustrate the eight problems and designers might have solved. Identification letters denote designers; they are placed in the figure to indicate which variant was solved by each designer.

TABLE 1. EIGHT VARIANTS OF THE SCHEDULING PROBLEM

Variable Aspect			Designer	Report Section
Priority	Frames	Constraint Failures		
External	Yes	Probable	D	3.3.4
External	Yes	Improbable	C	3.3.3.
External	No	Probable		
External	No	Improbable		
Internal	Yes	Probable		
Internal	Yes	Improbable		
Internal	No	Probable		
Internal	No	Improbable	A B	3.3.1 3.3.2

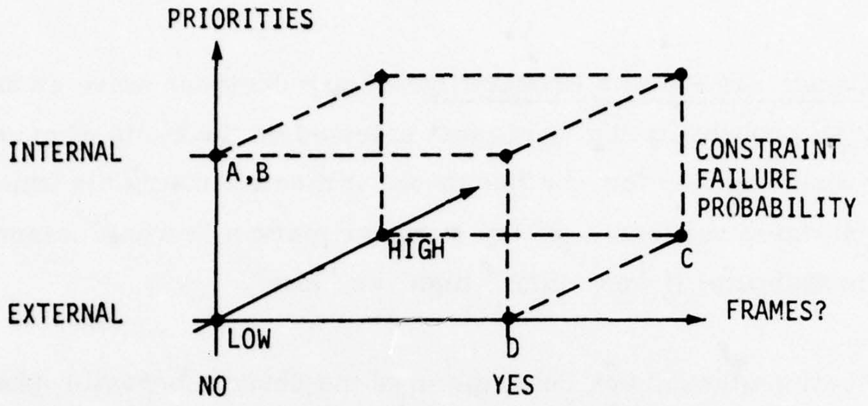


Figure 17. The Radar Scheduling Problem Space

3.1.1.1 Priorities--To fit various extant scheduling algorithms into a common framework, the first designer introduced internally-generated request priorities depending upon time, request parameter values, and request parameter rankings. Changing the priority function could change the algorithm to one of the extant algorithms. This specification was not clearly communicated to the other designers; the last two assumed that the request priorities were externally-fixed quantities that arrived with the requests. Thus, the two positions regarding priority are that it is externally or internally specified. Any architecture that works with internal priorities can be modified to work on external priorities; the reverse modification, however, is not simple.

3.1.1.2 Frames--Frame scheduling algorithms arbitrarily divide time into "frames" which are fixed length segments, each one scheduled as a block. The idea is that better schedules will be developed if the time horizon is limited; requests that do not happen to get into the schedule in one frame await the next frame. In the absence of frames, the schedule is constructed incrementally on a request-by-request basis.

3.1.1.3 <u>Constraint Failure Probability</u>--Every designer made an assumption regarding the probability that a request selected on the basis of priority would fail when checked for whether or not it meets constraints imposed by already-scheduled requests. Every designer made a "strong" assumption about his probability; it was either "high" or "low".

All specifications lacked any description of the desired behavior whenever the system saturated, that is, when the requests arrived more quickly than they could be served by the radar. Some architects made the explicit assumption that requests were never cancelled. If they allocated a fixed space to store requests, then their scheduler would saturate with the system.

3.2 PERFORMANCE OF CONVENTIONAL ARCHITECTURES

Before discussing novel architectures developed to efficiently solve the radar scheduling problem, we evaluate the advantages and disadvantages of several existing architectures when used to solve the scheduling problem. To make these analyses, we made preliminary design decisions concerning how the scheduling function might be implemented on each architecuture. We always tried to select a good design for evaluation. However, there may be other designs which do not suffer the same disadvantages or offer the same advantages as the one studied.

In the following sections, we discuss the assumptions along with the advantages and disadvantages of each particular system. These systems are covered:

1. Von Neumann uniprocessor systems
2. PEPE
3. Array machines
4. Pipelined machines
5. Multiprocessor systems

We are unable to quantify any comparisons because we have no quantification of the computational and memory requirements of the scheduling algorithm.

3.2.1 Von Neumann Uniprocessors

The major disadvantage of uniprocessors is their limited throughput. It does not seem that the throughput will increase sufficiently in the near future to generate good radar schedules. Furthermore, the conventional Von Neumann structure has many critical resources; if these failed, system failures would result.

3.2.2 PEPE

The PEPE machine[24] is a SIMD machine[25] which uses a set of three central control units (for three different classes of operations) and an ensemble of processing elements. For radar scheduling, we allocate each radar request to a separate processing element. At first glance, PEPE seems ideal for radar scheduling; common operations could be performed in the control units and request-specific operation sequences would be delegated to the processing elements. However, PEPE's single instruction stream may be a bottleneck for processing radar requests for the following reasons:

- Different request classes may require different handling before scheduling.
- Different constraints do require different checks.
- Different phases of the scheduling algorithm will have to be executed sequentially.

Request class differences are small (in terms of handling, but not in terms of scheduling difficulties), but constraint processing differences are large. Each constraint check requires a unique algorithm to relate a specific characteristic of a tentative schedule against a candidate request. A single instruction stream forces one to a design in which constraint checking is a very important consumer of processor time.

The third bottleneck becomes important when requests arrive after a schedule including many previous requests has been established. If the scheduling machine could execute multiple instruction streams, it could complete the schedule while receiving the new request (after which it may decide to discard the previous schedule).

Two other features of PEPE are the absence of data paths among the processing elements, and its correlation unit which, among other operations, is designed to find the maximum value among the values stored in the processing elements. The maximum selection mechanism can be used to find the highest priority request, directly helping the scheduling function; the absence of data paths slightly hinders radar scheduling since request parameter value rankings can be determined only through repeated maximum searches.

3.2.3 Array Machines

Array machines are constructed from a set of interconnected processing elements driven by a common control unit.[26] Thus array machines are similar to ensemble machines, such as PEPE. As in PEPE, we would assign separate radar requests to separate processing elements when we implemented radar scheduling.

Array machines differ from PEPE in two ways. First, there is no special correlation, associative, or maximum-value circuitry connected to the processing elements. Second, the array contains data paths connecting the processing elements into an array (recall that PEPE had no such connections).

We did not, however, find any effective way to use the array interconnections to assist with the radar scheduling problem. We can use the array as a sorting vector to determine parameter value rankings, that is, which operation is useful for computing request priorities. In fact, the architecture developed by designer A incorporates an array of (simple) elements to sort requests in this manner (see Section 3.3.1.3 for details).

The array machine has the same disadvantages as PEPE due to the single instruction stream issued by the control unit. Furthermore, the array cannot function if any single processing element fails. Therefore, the array offers no significant advantage over the ensemble for radar scheduling.

3.2.4 Pipelined Machines

We consider pipelined machines[27] with vector instructions, such as TI-ASC and CDC STAR-100. In most logical respects, these machines are similar to array and ensemble machines: there is a single instruction stream operating on a vector of data. By placing radar requests into vectors, they can be handled very easily. The nonuniformity of the constraint processing problem again produces inefficiency since each constraint requires a different program.

When used for radar scheduling, vector machines also suffer from the major disadvantage of the single instruction stream.

3.2.5 Multiprocessors

A multiprocessor system[28] overcomes the major disadvantage of the preceding systems: it can execute multiple instruction streams. Multiprocessor systems can be classified by the "degree" of coupling among the processors and the techniques used to achieve it. Tightly-coupled multiprocessors have high bandwidth coupling, typically through shared memory or shared data paths (buses, etc.). Loosely-coupled multipossessors have only low bandwidth interconnections, typically through input/output channels.

Low bandwidth interconnections appear to be inadequate for the radar scheduling problem because interconnections are used on every constraint check for every request scheduled. This statement assumes that the processor multiplicity is used to solve the constraint processing problem that permeated the difficulties with uniprocessor systems.

The conclusion is that constraint processing, especially, requires multiprocessors with high-bandwidth interconnections. Our distributed processing systems use such interconnections.

3.3 FOUR DESIGNERS

In the following four subsections, we describe the efforts of four designers of different backgrounds and experience as they developed radar scheduling architectures. We present the efforts of the first designer. The second designer chose to work from the first designer's results to create modified architectures. After these efforts, we asked two designers to develop architectures and explicit methodologies for architecture development.

The efforts of the designers are compared in Section 3.4.

3.3.1 Designer A's Architecture

Designer A was the first designer of a radar scheduling architecture; he produced a description of the radar scheduling algorithm, described in Section 3.3.1.1. He was not asked to develop a methodology, so his development followed ad hoc reasoning, summarized in Section 3.3.1.2. How the architecture is used to implement radar scheduling is discussed in Section 3.3.1.3.

3.3.1.1 The Problem--As noted in Section 3.1, the designers individually interpreted the scheduling algorithm description. Designer A developed an architecture to solve the scheduling problem without frames, with a low conflict probability and with internal priorities. His design process, described in Section 3.3.1.2, produced the architecture of Section 3.3.1.3 to solve this problem.

3.3.1.2 The Architectural Development Process--The architecture was designed entirely heuristically in two phases. The first phase was to structure the scheduling function so as to reveal potential concurrency. Then an architecture was devised which modeled that structure. The second phase was to identify inefficiencies and combine dedicated data paths into multipurpose paths.

3.3.1.3 Architecture--We describe the architecture in three subsections. After describing the processing elements and their functions (Section 3.3.1.3.1) and the communications paths (Section 3.3.1.3.2), we turn to the operating philosophy and the control strategy in Section 3.3.1.3.3.

Figure 18 shows the complete architecture. The designer did not design for reliability; thus in the following discussion we ignore reliability questions.

3.3.1.3.1 Processing Elements--Each processing element contains memory. There are five classes of processing elements:

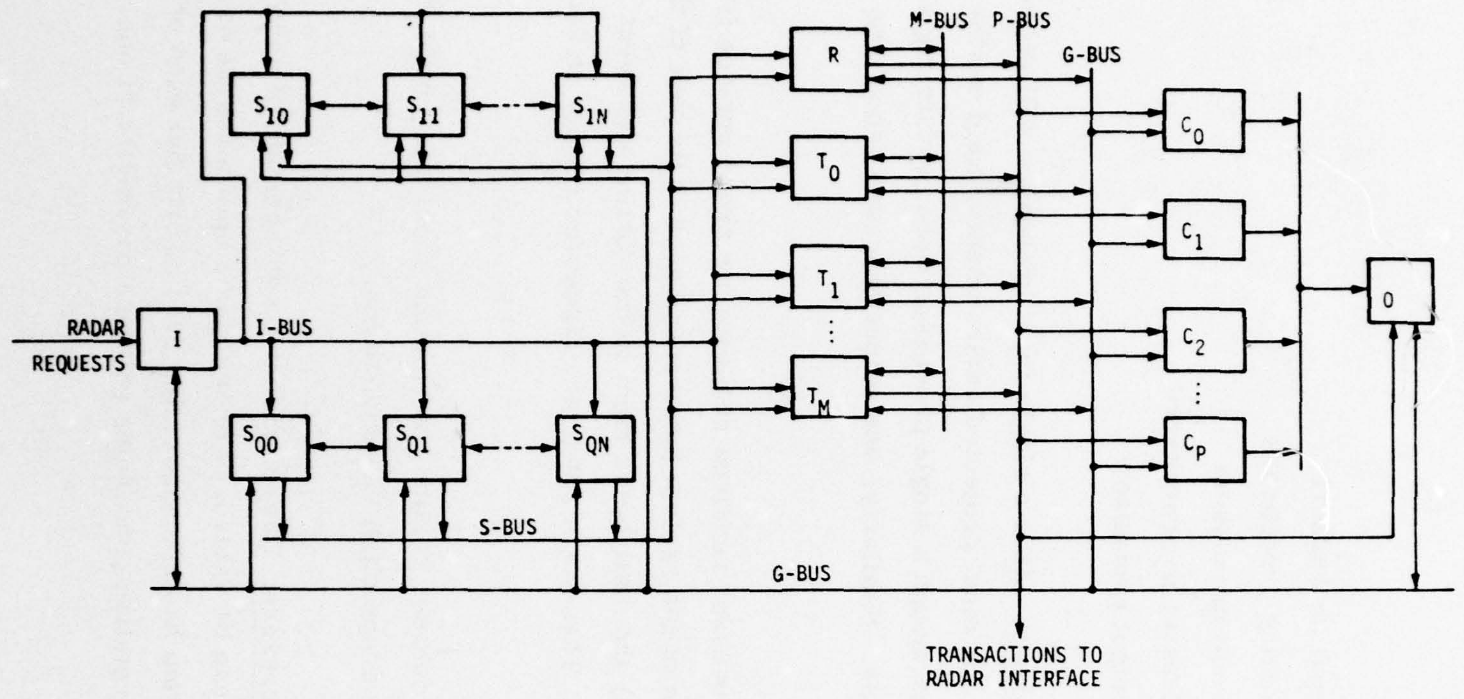


Figure 18. Designer A's Radar Scheduling Architecture

1. Input processors
2. Sorting processors
3. Event processors
4. Constraint processors
5. Output processors

Input Processors: Since a single input processor will probably be sufficient for most projected environments, we describe the processing as though a single processor were performing all the input functions. Similarly, one output processor should be adequate.

The input processor receives requests for radar service from higher levels of the system (here considered to be part of the environment) and distributes them to the sorting and event processors. These operations are discussed further in Section 3.3.1.3.3.

The input processor retains information about event processor allocation to accomplish these functions.

Sorting Processors: The number and arrangement of sorting processors can be varied. For discussion purposes we assume that the system has enough sorting processors that each one handles information concerning only one parameter of one request.

The sorting processors are arranged in a rectangular array. The processors in one row sort the requests according to one request parameter. Whenever a new request arrives at the scheduler, the input processor commands the sorting array to insert the new request into the sorted lists. This operation is the primary function of the (simple) sorting processors. However, each also must be able to recognize the identity of the request it holds and to delete the request, closing up the sorted list accordingly, upon command from the input processor.

Event Processors: There are two types of event processors: transmit event processors and a receive event processor. Since the receive event processor is not required to perform many computations, a single processing unit will be adequate (ignoring reliability questions). For discussion, we assume that each transmit event processor holds information concerning a single request. These processors evaluate the priority function, compare the resulting values with other transmit event processors using the M-BUS (see Section 3.3.1.3.2), and update request parameter rankings whenever a request is scheduled or a new request enters the system.

Constraint Processors: The constraint processors check the request that is a candidate for scheduling (by virtue of its high priority) to determine whether that request would violate any radar constraints if scheduled. Each constraint processor performs a different algorithm to check a different constraint. The constraint processors must retain information concerning the recent radar schedule as necessary to evaluate the constraints and determine whether they have been violated.

Output Processor: Generally the output processor controls priority determination and constraint checking. When a request has high priority and also meets the constraints, it is scheduled. The output processor handles management details to insure that these checks are made properly.

3.3.1.3.2 Communications Paths--Five communications paths connect the processing elements in the system: the I, S, M, P, and G buses. In this description we suggest which processor might control each bus if centralized bus control is desired. The designer felt that bus control details do not belong at this level of design, so no bus control protocols or policies are discussed here.

I-Bus: The I-bus carries requests from the input processor to the sorting and event processors. The input processor controls use of this bus.

S-Bus: The S-bus carries request parameter rankings from the sorting array to the event processors where they are stored. This bus is controlled by the destination event processor (which had been selected by the input processor).

M-Bus: The M-bus is used exclusively for maximum priority selection and for conflict resolution whenever several transmit requests have identical priorities. The receive processor controls this bus.

P-Bus: The P-bus carries the parameters of the selected request to the conflict processors and the radar (when the request is to be scheduled). This bus is controlled by the output processor.

G-Bus: The G-bus carries global control signals that synchronize the system. The input processor controls this bus.

3.3.1.3.3 System Operation--The system schedules all requests that it knows about until either they all have been scheduled or a new request arrives. Scheduled requests are held in event processors until they are sent to the radar. The schedule is a queue of pointers to these request descriptions; the pointers are held in the output processor. When a new request arrives, the existing schedule is discarded (except for some requests at the head of the queue that are retained to keep the radar busy during rescheduling), and a new schedule is developed serially.

Events that cause interesting computations include:

- A new request arrives.
- Request priorities are compared.
- A request fails constraint checks.
- A request is scheduled.
- The radar removes a request for servicing.

New Request Arrival: In addition to the possible schedule cancellation mentioned above and described below, the arrival prompts a set of sorting operations, an update of the parameter rankings of all requests, the assignment of a transmit event processor to the event, and the copying of all information concerning that event to the selected processor. The input processor controls these processes although the output processor must control any schedule manipulations.

Every parameter requiring a rank value for the priority function is assigned to a row of sorting processors (one row of processors may sort for several parameters). Each sorting processor holds the value of the parameter for an active request; it compares this value against the value of the incoming request's parameter. If the retained value is less than the incoming value, the retained value is copied to the neighbor further away from the event processors (the highest parameter values are held nearest the event processors). The processor which sends its own value out but does not receive one from its neighbor is the processor to receive the incoming value. This processor will transmit the ranking of the new request on the S-bus; every row of sorting processors has a private set of lines to send this information. Sorting assigns a rank to the new request but it does not update the ranks of the requests still held in event processors. This updating occurs when the sorter sends the new rank across the S-bus. Every event processor listens to all rank transmissions on the

S-bus; if the new event is ahead of the retained event, then the rank of the retained event must be adjusted by 1 (whether up or down depends on the encoding of ranks).

The new arrival must find space in a transmit event processor where its parameters will be stored. The input processor is in charge of this space management function; it maintains a table of transmit event processor status and selects a free processor to receive the new request. The identification of the selected processor is transmitted on the G-bus; when a transmit event processor detects its number on the bus, it prepares to receive a new request. The receive event processor copies the parameters of every request for its internal use.

Now the system is ready to cancel the existing schedule and begin rescheduling. Some of the existing schedule may be preserved to guarantee that the radar is not idled simply because the scheduler is busy reworking the schedule.

Compare Priorities: Priority comparison is one important step in schedule determination. Requests that have been scheduled do not participate in the priority comparison; all unscheduled requests will be called "active." Before the priorities can be compared, they are evaluated by the transmit event processors. The details of this evaluation are beyond the scope of this study. Each transmit event processor holding several active events selects the active event with the highest priority as its candidate for the next spot in the schedule. The candidates are compared by bit-serial transmission on the M-bus. Two "wired-OR" signals in this bus perform the comparison. One receives information bits, while the other is held at 1 until the information bit has been transmitted. Each active processor examines the bit on the information bus; if that bit is "1" and if the processor transmitted a zero, then the processor drops out of the competition. Ties can be broken by transmitting the processor number after the priority value.

The processor remaining as a competitor at the end of the bit sequence is the one holding the highest priority request in the system. Next that request is checked against the radar constraints to determine whether it can be scheduled.

Constraint Check Failure: A request which fails constraint checks cannot be scheduled. Another priority comparison, with the failing request not participating, is required. The transmit event processors must manage this nonparticipation properly; when a request is placed in the schedule, any request that failed to make the schedule due to constraint failures must be reactivated for the next cycle.

Request Scheduled: When a request passes the constraint check, it is entered into the schedule. The output processor enters a pointer to the event processor holding the request (and its identification within that processor if the processor can hold several requests) into a queue within its memory. The request becomes inactive in further priority comparisons (unless the schedule is discarded due to a new arrival). The receive interval corresponding to the selected request is scheduled, and the constraint processor that checks for radar usage adds the receive interval to its checking.

Request Sent to Radar: When a request is actually sent to the radar, it must be deleted from the scheduler; to do this, the request must have sufficient identification. Information about the request is stored directly in one transmit event processor, one receive event processor, and one sorting processor in each row of the sorting array. Implicit information is contained in the rankings of all other requests being scheduled.

The information in the transmit event processor is easily deleted since the schedule queue contains a pointer to that information. Receive event processor memory space could be allocated in a fixed way related to the transmit event processor allocation, so this information also can be deleted when the pointer is known.

The remaining information all relates to the ranking of the performed request. If the output processor broadcasts these rankings on the G-bus, proper updating is easily implemented. For each rank parameter broadcast, two actions must be taken: 1) the sorting processor holding the request must delete it; lower ranked-entries move up to fill the gap; and 2) the ranks of all requests must be adjusted to account for the missing rank. The latter can be achieved by the inverse of the rank adjustment process used upon receipt of a new request.

The receipt of the echoes resulting from the transmission must be scheduled. Any schedule cancellation cannot affect these receive events which must be marked to ensure that they are scheduled.

3.3.1.3.4 Observations--This design widely distributes the functions; a large number of modules are specialized to perform only one function. Compressions of modules into larger modules are compatible with the overall bus structure (which is the essence of the architecture). The next designer used these opportunities to produce his architectures.

This design represents an algorithm compromise: Since the optimum schedule is hard to find (because many combinations must be examined), the priority-class algorithms were selected instead. Furthermore, the priority computations do not receive information concerning whether the request ever failed constraint checking. Such information might be useful in improving the performance of the algorithm.

3.3.2 Designer B's Architecture

Designer B also was not asked to develop a methodology. He chose to work from the first architecture, producing variants by finding important subfunctions whose performance could be improved. His design process is summarized and his architectures described in Section 3.3.2.2. First we discuss this designer's conception of the scheduling problem.

3.3.2.1 The Problem--Designer B recognized two dimensions of the scheduling problem. He discussed both frames and the issue of whether requests should be processed priority first or constraint first, and he pointed out that most architectures will reflect this choice.

On the frame issue, Designer B noted that frames are not a necessary part of the scheduling algorithm although frames make the scheduling problem easier. But this ease is gained at a price: not all requests may be considered for every slot in the schedule (this is based on one scheduling strategy in which requests in the same class--track, scan, or identify--are gathered into the same frame). This restriction can reduce the value of the schedule produced by the algorithm.

The issue of whether priorities or constraints are processed first reduces to the issue of estimating constraint conflict probabilities, for it may be more efficient to select the strategy that removes the largest number of requests in the first stage so that second stage processing is simplified. If there is a high probability of constraint violation, one should consider designs in which the requests are winnowed by constraint checking before the survivors are compared by priorities. If, however, there is little chance of constraint violation, the priorities should be checked first. It might be difficult to perform constraint checking on a set of requests for a sequence of schedule positions due to the combinational interactions. Designer B assumed that the constraint failure probability was low.

Thus, his architecture embodies internal priorities, does not use frames, and assumes constraint failures to be improbable, as indicated in Table 1.

3.3.2.2 Design Technique and Architectures--Designer B's technique was to successively improve an existing architecture, based upon introspection of the architecture. The designer looked for problem areas or points where he thought the architecture could be improved. For each area he developed alternate realizations of the subfunction, usually increasing the speed or lowering the cost. Each alternate realization was considered in combination with alternate designs of other subfunctions to identify and resolve conflicts and thereby realize the next architecture in the sequence. If more bottlenecks were apparent in the new structure, the process was repeated.

Since this technique is so intimately connected to a specific architecture and how it is used for scheduling, we discuss the design stages and the architecture together in this section.

Designer B first studied constraint processing. He reasoned that, if the constraint violation frequency were high enough, repeated sequential constraint processing would be needed to find an acceptable request; this would be less efficient than parallel checking of the constraints. Therefore he moved the responsibility for constraint checking to the transmit event processors. This change increased the scheduling rate; the highest priority request in every event processor was checked against the constraints in parallel, and then the survivors participated in a maximum priority comparison to select the next scheduled event.

Second, the designer noted that the event processors are idle when the sorting processors are inserting a new request into the array. Since the sorting function is not complex, he moved it to the event processors. Each transmit processor must have access to all parameters of all requests to perform this sorting for the new request; all old requests can compute their new rankings by simple comparisons, as discussed in Section 3. 3. 1. 3. After this change, the system includes only event processors and the input and output controller processors, as shown in Figure 19. The buses in this structure have functions similar to the buses in Designer A's architecture (see Section 3. 3. 1. 3).

Next the designer improved the reliability by adding redundancy. He also separated each transmit processor into two separate units: one to perform sorting and the other to deal with the maximum priority determination on the M-bus. He did not specify which of the two would compute the priorities or check radar constraints.

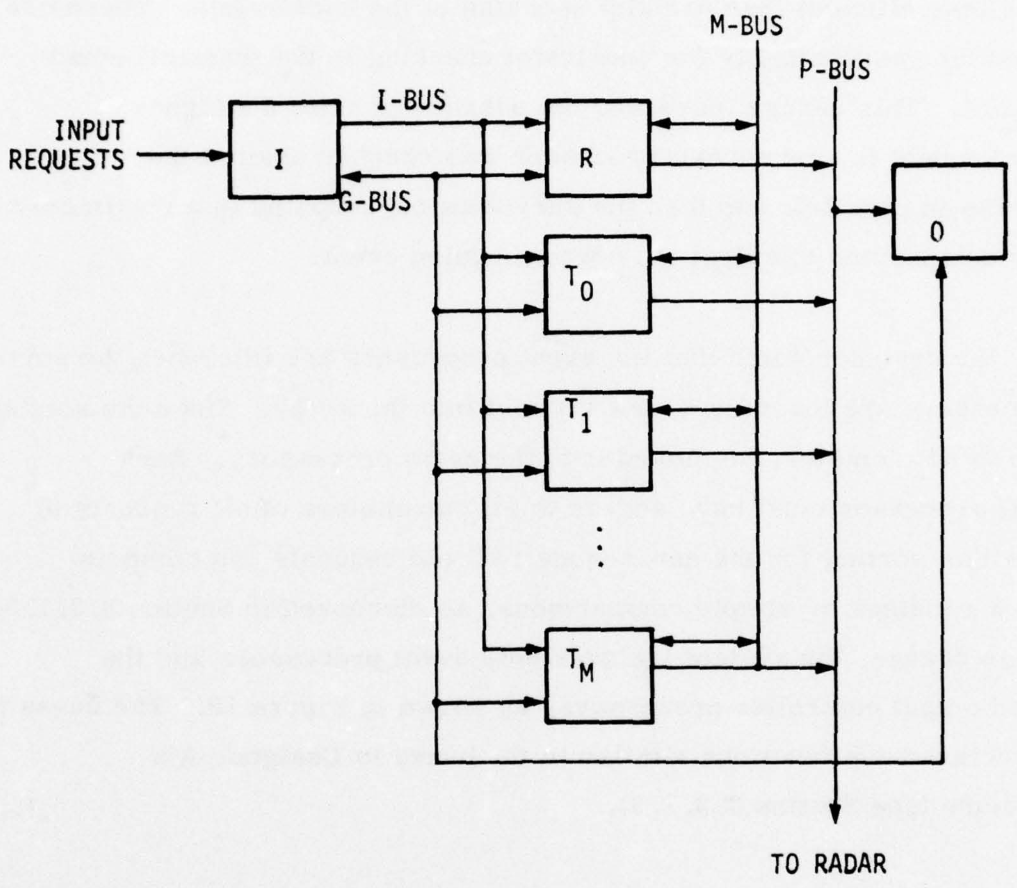


Figure 19. Designer B's First Architecture

The designer added more redundancy and specified that the centralized function processors (I, O, and R) be identical so they could perform resource allocation among themselves and achieve graceful degradation and redundancy at minimal hardware cost. Figure 20 illustrates an architecture meeting these conditions, where C denotes a "centralized function" processor.

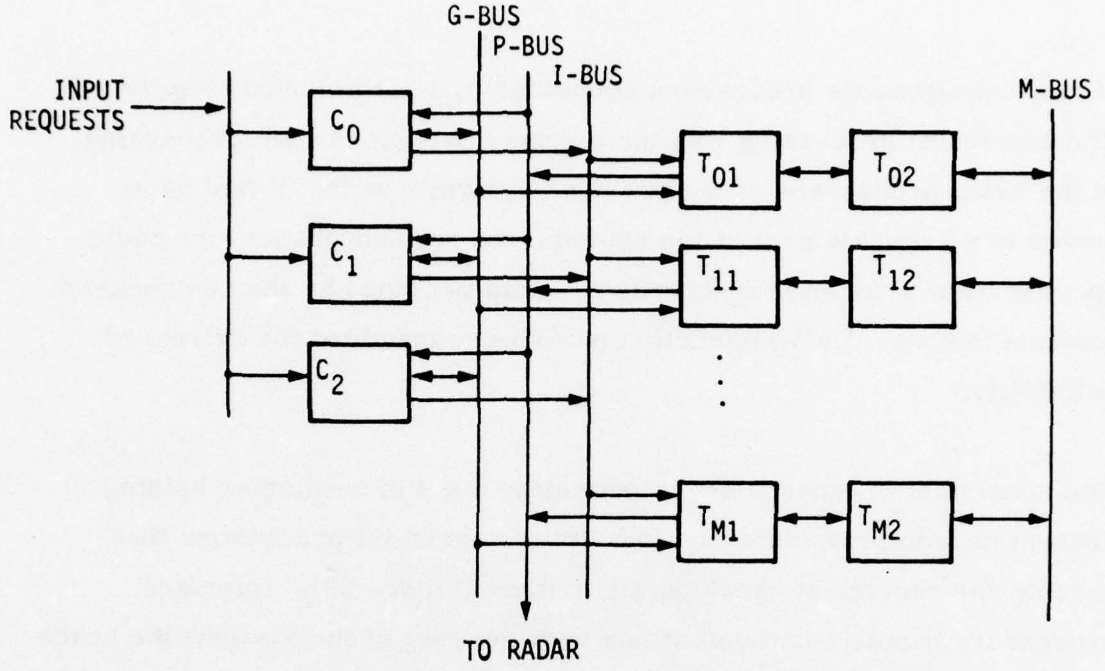


Figure 20. Modularized Architecture

Designer B later reconsidered his previous design and developed another one. His first observation was that constraint checking required a lot of processing power, which is not effectively used since it is used on all requests without regard for their priorities. He wanted to use the available constraint processing power more efficiently without going back to sequential constraint processing. Also, he wished to retain the reliability and redundancy of the previous design.

The designer's solution to this dilemma was to provide a "pool" of constraint processors coupled to the event processors through a set of shared memory modules. At the highest level, the system appears as a set of non-homogeneous processors connected to a set of buses (Figure 21). The constraint processing was the primary data-dependent processing in the event processors of the previous design; now that it had been moved to a separate part of the system, the request processors could operate from a common instruction stream provided by the centrailized function module. Dual instruction paths were provided for increased reliability.

The constraint processor set is primarily a set of memories holding request parameters connected to a set of constraint processors that execute the constraint checking algorithms (Figure 22). Interface processors handle communications with the rest of the system; the boxes labeled "constraint algorithms" are memories holding the constraint checking programs. (The designer did not specify where the radar status was stored.) Each constraint processor is programmed to check only a subset of the set of constraints; the designer suggests selecting the subsets on a cyclic basis.

Management of the "distributed partial interconnection" between the request parameter memories and the constraint processors is the key to the system's operation. Request parameters arrive along the request parameter path in priority order. Idle constraint processors scan the request memories for unprocessed constraint checks. (A status vector with each request handles the bookkeeping.) Upon finding work, the constraint processor proceeds to check the constraint for the new request. When it has finished the check, it updates the status vector. If all constraints have been checked and the request has passed all of the checks, the processor signals that this request should be scheduled next.

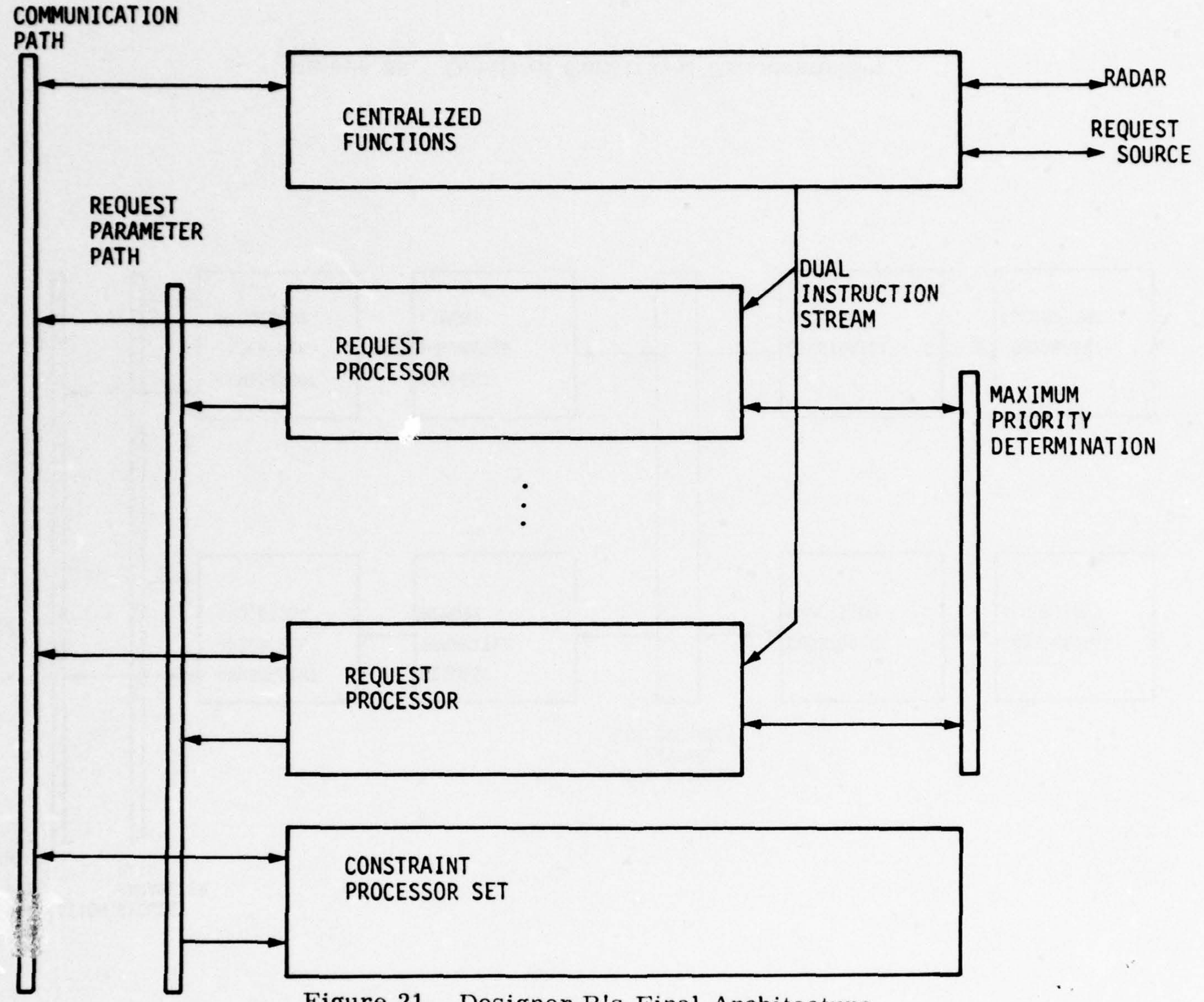


Figure 21. Designer B's Final Architecture

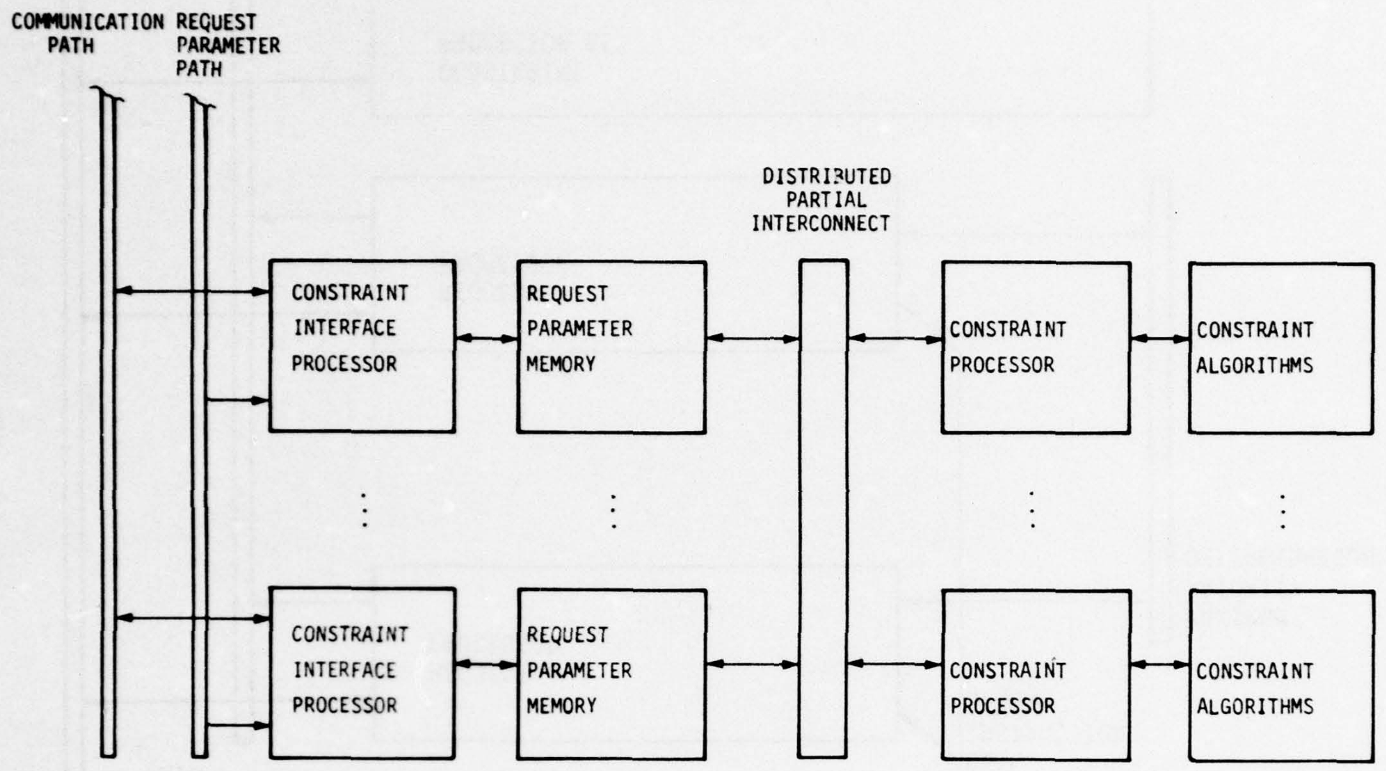


Figure 22. Detail of Constraint Processor Set

If the constraint check fails, the constraint processor should remove that request from consideration. (This seems to require a coordination mechanism not specified by the designer.)

The interconnection between constraint processors and request parameter memories need not be total, provided that every constraint can be checked for every request by some processor. Contention among the requests for constraint processors is resolved with a priority bias because the requests arrive in priority order, and those present earliest will command more constraint processing. Therefore the first successful request probably has a very high priority.

3.3.3 Designer C's Architecture

Designer C was asked to develop a design methodology, described in Section 2.1. He used this methodology to develop a radar scheduling architecture, described in Section 3.3.3.2. First we discuss his version of the scheduling problem.

3.3.3.1 The Problem--Figure 23 shows an outline of the family of radar scheduling functions considered by Designer C. Arriving requests are of different classes. Each request has its own external priority, called local priority. Requests of the same class are ordered according to their local priorities. The partially sorted requests are sent to the framing function F_2 to be allocated in time frames based on global priorities given to the system. Then the frames are sent, one by one, to the checking function F_3. Function F_3 examines each frame and decides whether or not this frame meets the constraints, as described in Section 3.1. Accepted

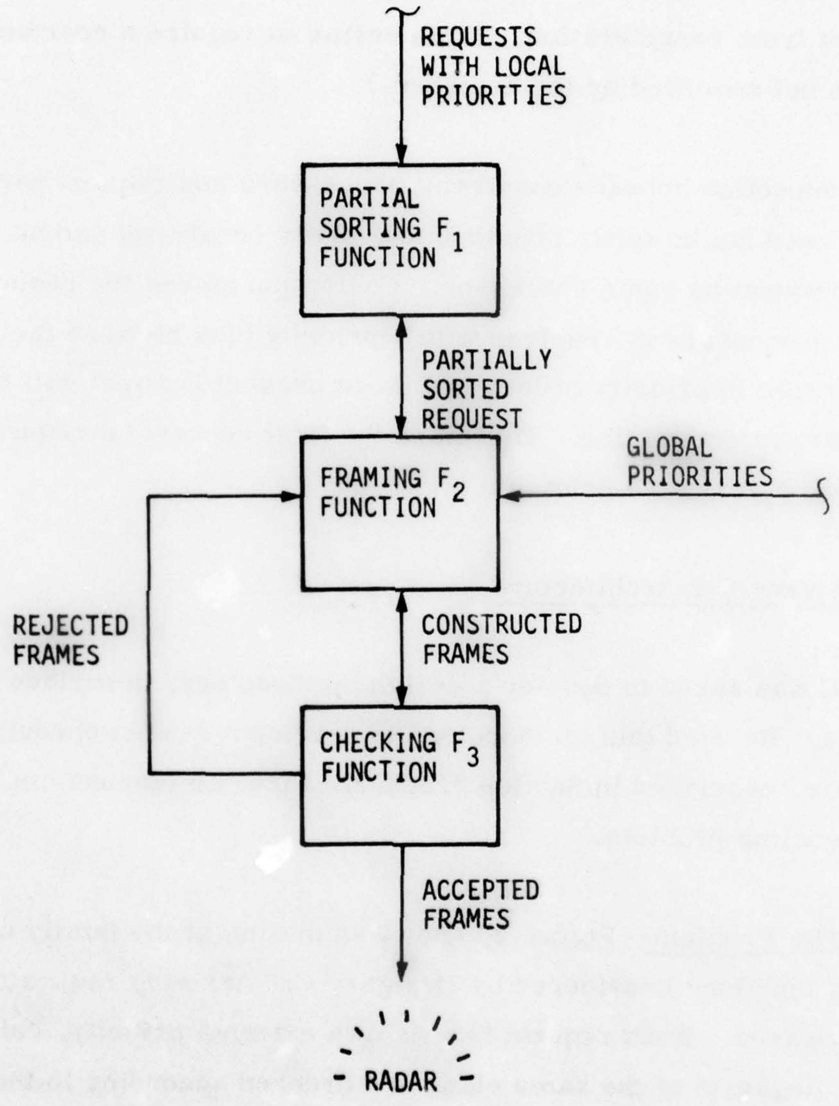


Figure 23. A Family of Radar Scheduling Functions

frames are sent to the radar, while rejected frames are sent back to F_2 to be corrected before being examined once more. This statement places Designer C's problem in the category having external priorities, using frames, and assuming a low constraint conflict probability.

3.3.3.2 Architectural Development--Recognizing some of the parallelism between operations, each of the two functions F_1 and F_3 can be decomposed into components, as shown in Figure 24. This particular decomposition is based on the assumption that there are three classes of requests (search requests, confirm requests, and track requests) and that there are two conditions to be examined for each frame.

The radar scheduling function after decomposition can be represented as an architecture graph (shown in Figure 25). If composition transformations are applied on this architecture graph, we can generate a large number of reduced graphs; each of them corresponds to a possible architecture. A number of these reduced graphs are shown in Figure 26 where nodes are labeled with numbers to help the reader follow the sequence of applying the composition transormations.

The resulting architectures are compared with respect to their cost, reliability, and other nonfunctional payoffs; the best architecture is then chosen.

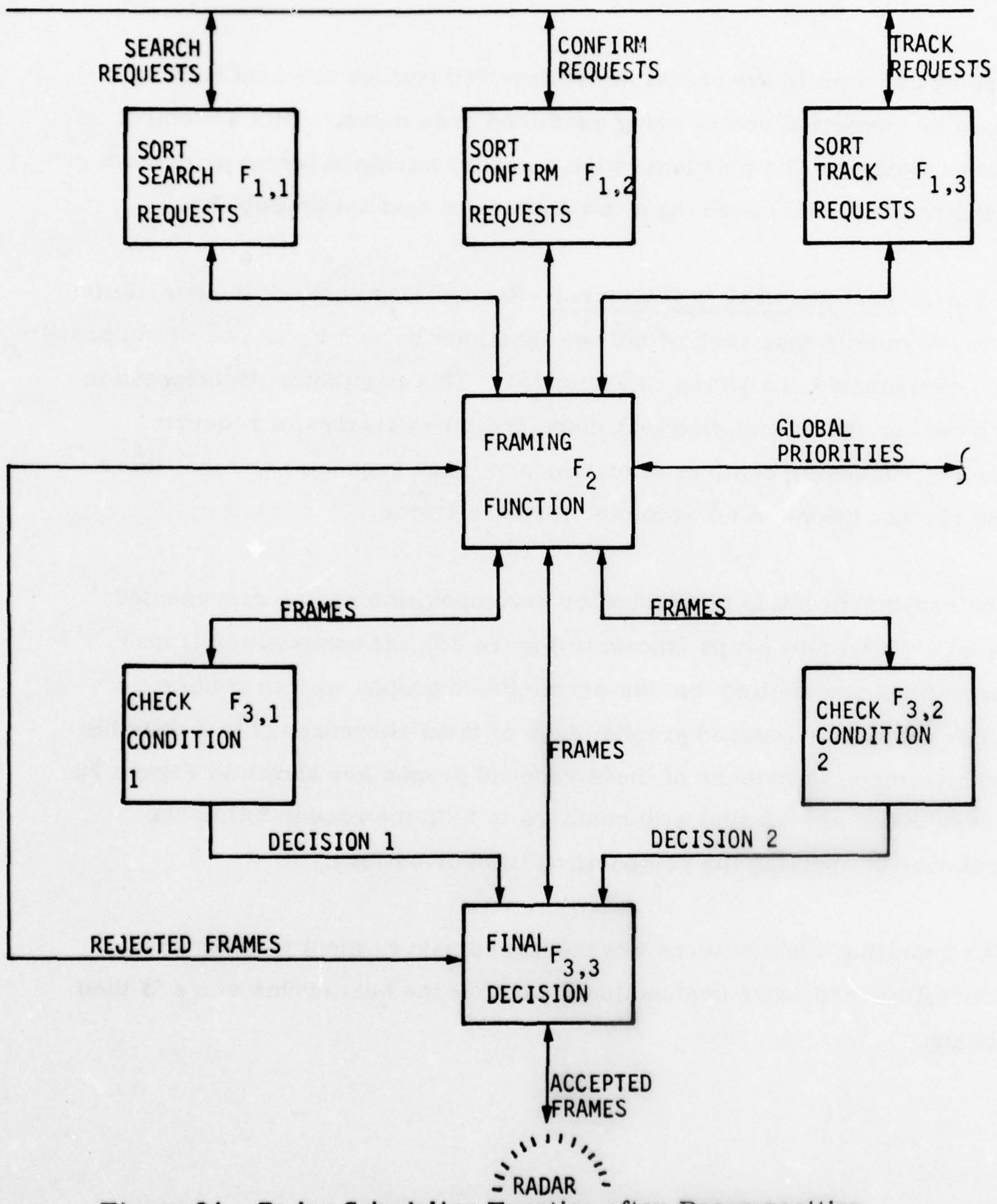


Figure 24. Radar Scheduling Function after Decomposition

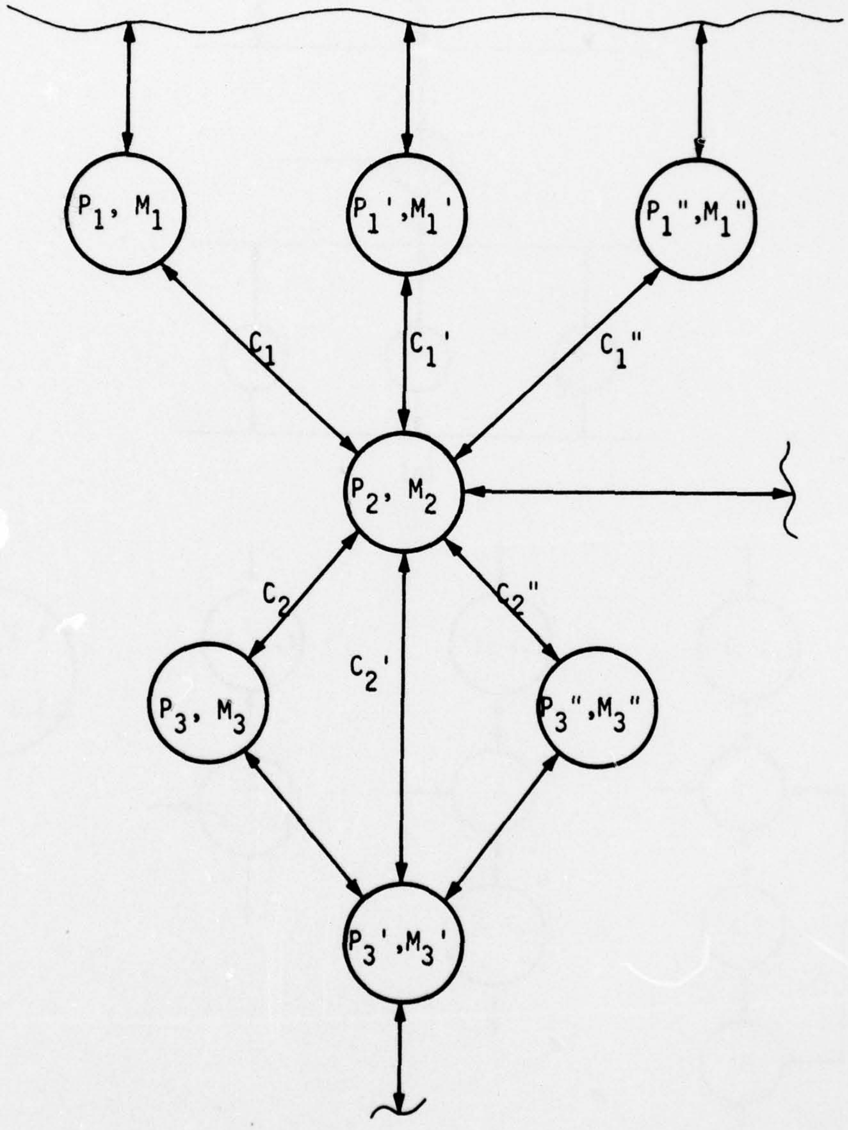


Figure 25. Architecture Graph for the Radar Scheduling Function in Figure 24

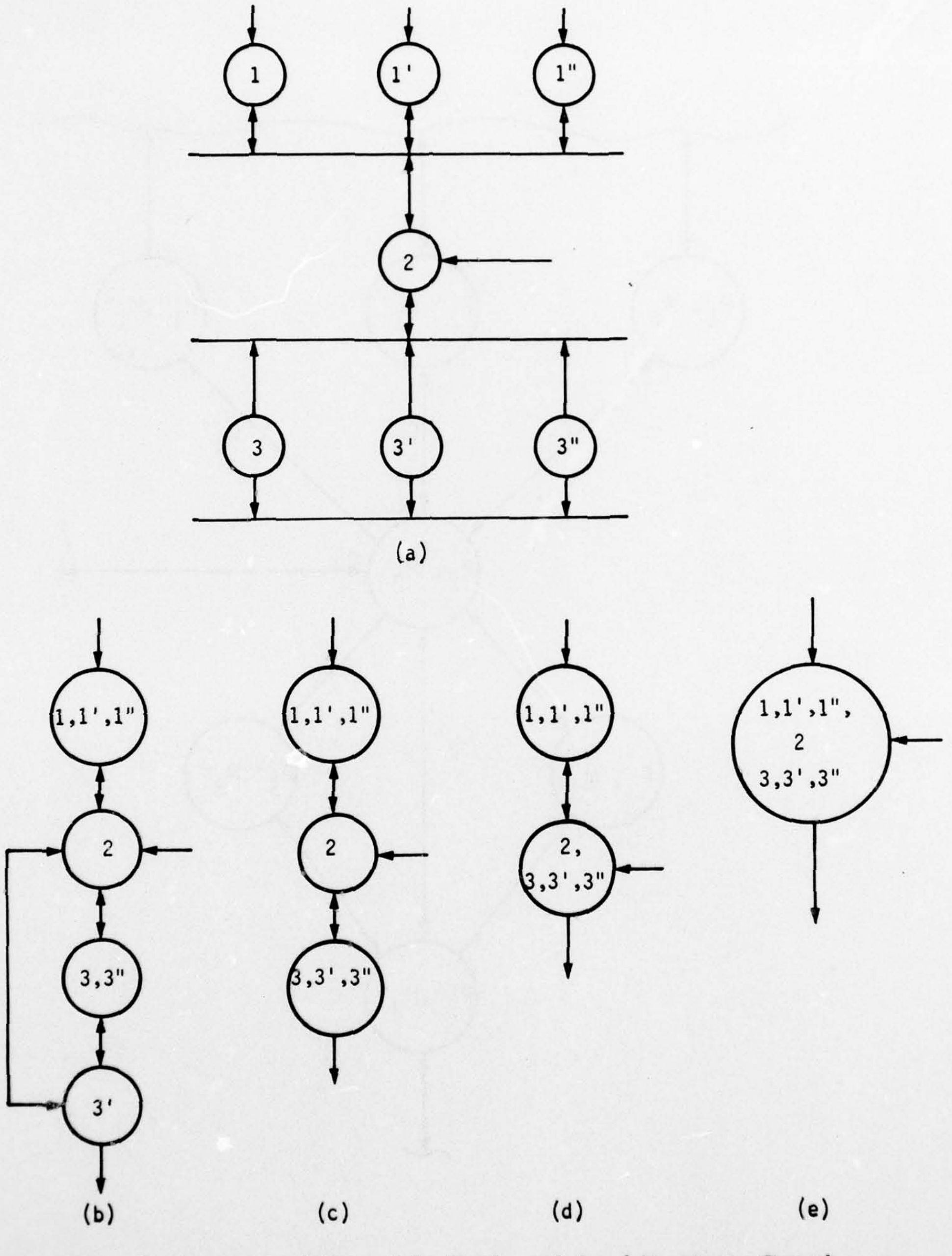


Figure 26. A Number of Possible Reduced Architecture Graphs (Each one corresponds to a system architecture.)

3.3.4 Designer D's Architecture

Designer D had developed the methodology described in Section 2.2 while designing an architecture to perform the radar scheduling function. We will describe his design with many references to the methodology.

This designer was asked for his perception of the problem, his design sequence, the assumptions that he made while doing the design, and how his architecture schedules radar requests. These issues are described in the following subsections.

3.3.4.1 The Problem--This designer viewed the radar scheduling function as follows:

- The input is a set of requests with associated (external) priorities, constraints on the scheduler, and the current radar status.
- The output will be a schedule of radar usage that will effectively satisfy all requests. Furthermore, the function will update the radar status.

Figure 27 illustrates a radar status time sequence.

This problem specification includes external priorities, frames, and a high conflict probability.

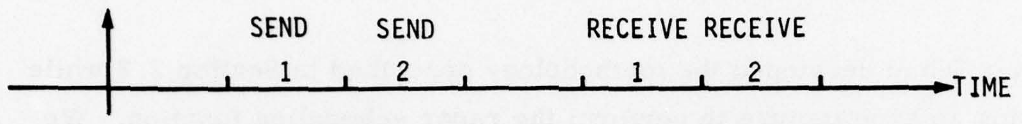


Figure 27. Radar Status Sequence

3.3.4.2 The Design Process--In this section, we describe the designer's development sequence and his assumptions regarding the problem that enter into the design decisions. Step numbers refer to the steps described in Section 2.2.

The designer, in Step 1, decomposed the scheduling function into three subfunctions:

1. Determine conflicts
2. Sort requests
3. Construct schedule

In the following description, the subfunctions are identified by these numbers. Each of these subfunctions requires information from several, possibly all, requests. The three subfunctions obey the precedence relations depicted in Figure 28 from which we note that subfunctions 1 and 2 could be performed in parallel.

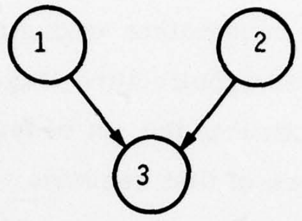


Figure 28. Subfunction Precedence Diagram

Step 3, developing a policy for handling a set of objects, is not required since the subfunctions are defined in terms of the whole set of requests. Detailed policies are considered while designing the architectures for subfunctions in Steps 6 and 11.

The designer selected the conflict determination subfunction for the initial implementation (Step 5). His architecture is based on the assumption that there will be many potential conflicts. It is also based on the following observations (made while attempting to uncover all possibilities for parallelism in implementing the subfunction):

- Feasible regions for individual requests can be found independently of the other requests.
- Relating the feasible regions of several requests may require more processing than that needed to find the feasible regions for each one.

The first assumption is not shared with other designs.

Having a method for relating constraints among requests, the designer, in Step 6, developed a distributed architecture (Figure 29) to implement that method. An "A" module constructs the set of feasible times for satisfying a request from the parameters of that request. Each "B" module relates three* feasible time schedules to produce a single composite feasible time schedule. The resulting schedule is then presented to the next module in the tree (unless, of course, it is the overall schedule).

Now the designer turned to reliability improvement problems. Noting that the modules nearer the tree root were more important for reliability, the designer studied their interconnection first. He decided to replace the last two levels of the tree with a set of identical processors connected between two buses, as shown in Figure 30. In this interconnection, extra processors can easily substitute for any failed processor.

The redesigned tree root was so attractive as a solution to the reliability problem that the designer replaced each level of the tree with a single set of processors sharing the same set of buses, as shown in Figure 31. Each processing element in this diagram contains an associated memory unit (not explicitly shown). Noting that the increased flexibility was helpful, the designer then merged all sets of processors into a single set of identical processors (Figure 32); this completed his redesign for reliability.

* This number is a design parameter to be determined by implementers.

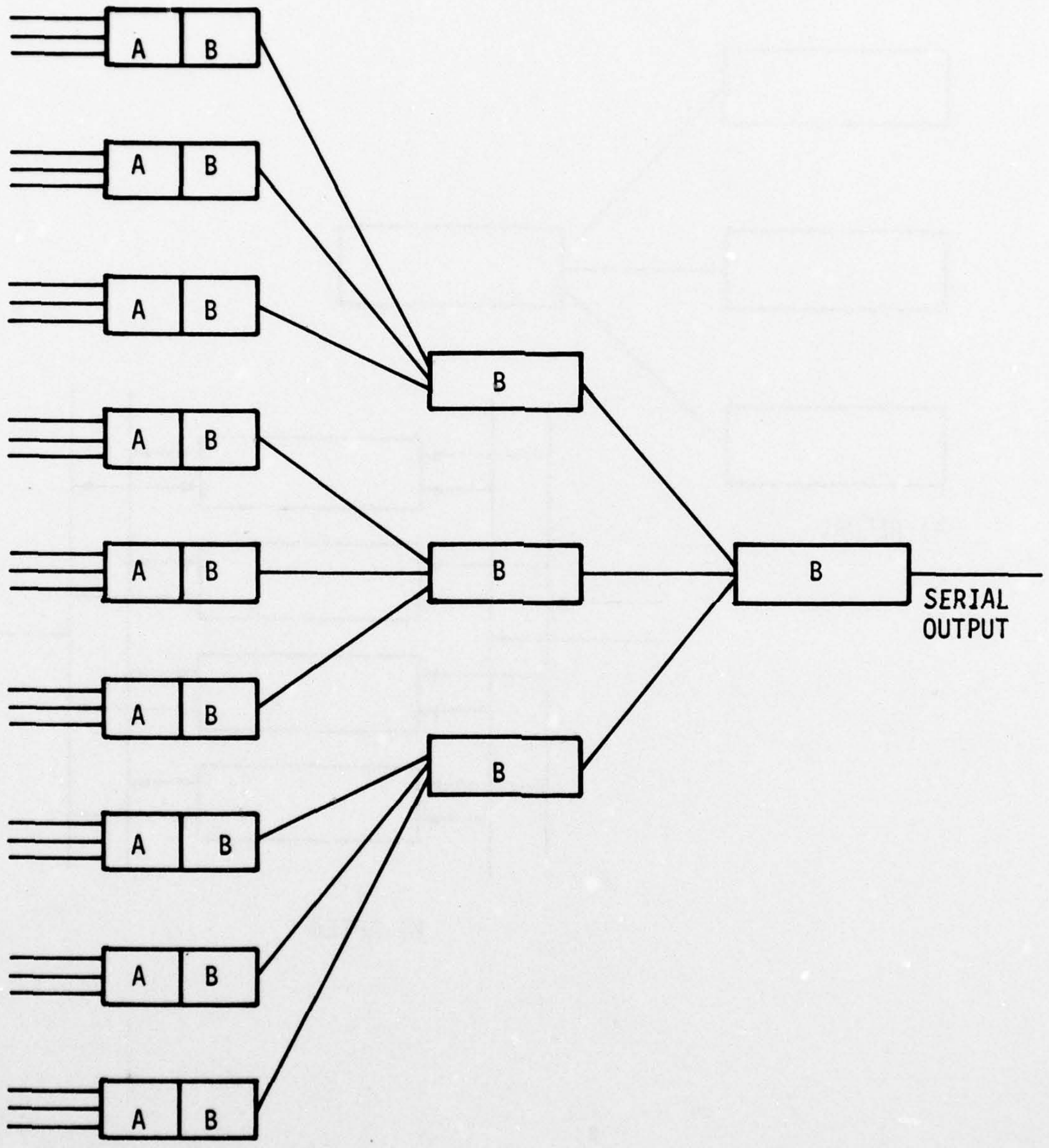


Figure 29. Conflict Processing Architecture

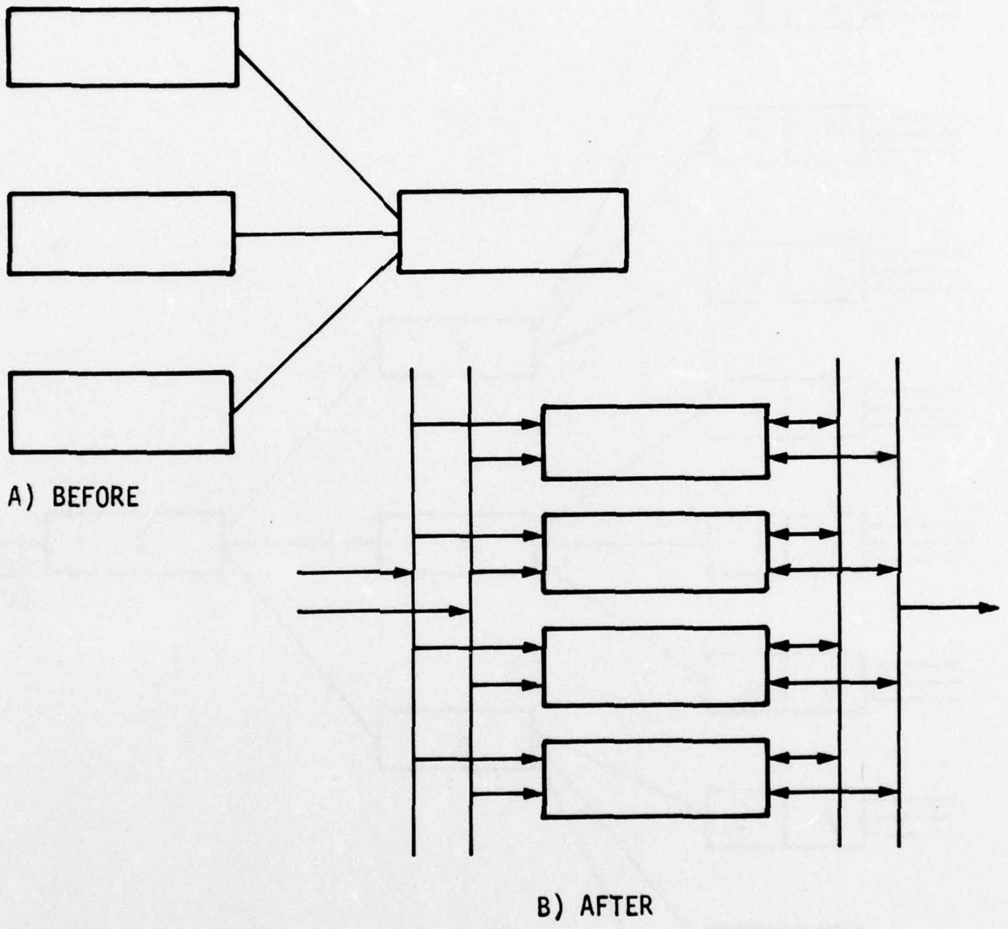


Figure 30. Tree Restructuring

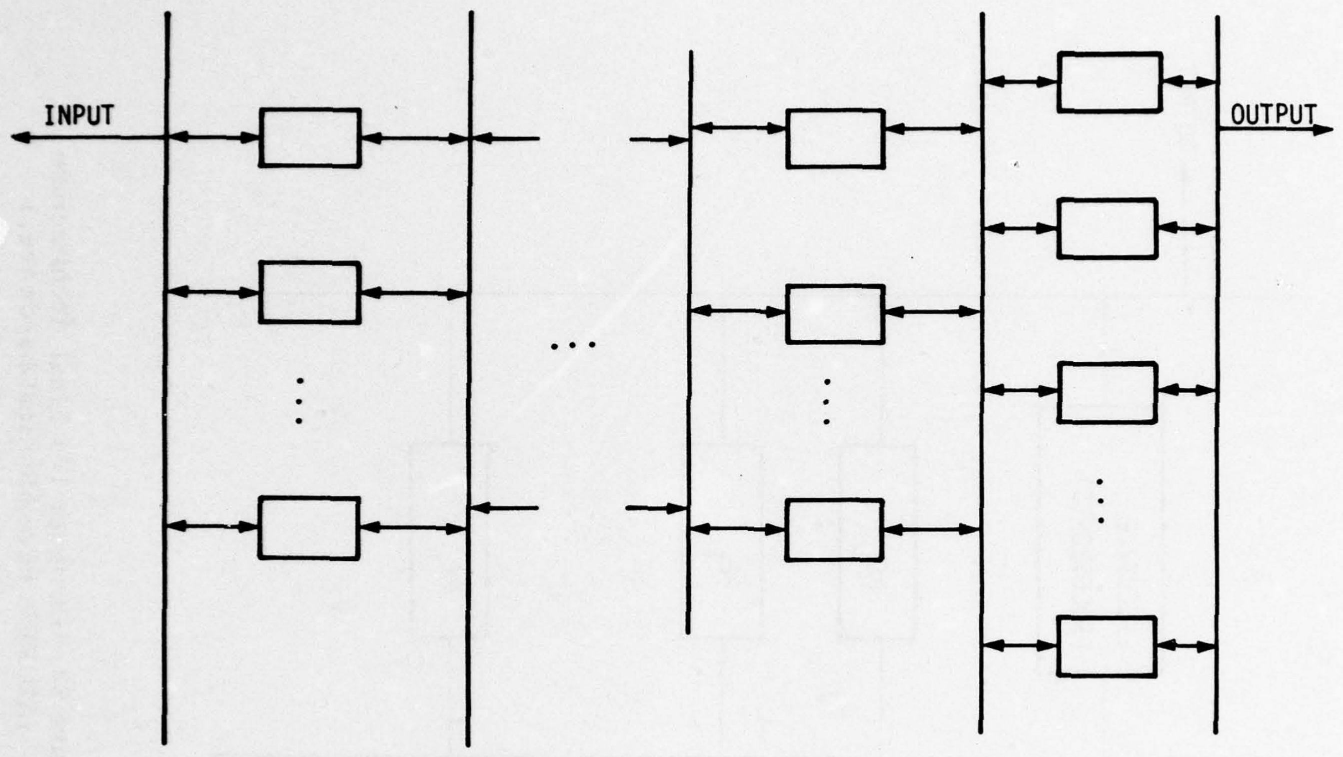


Figure 31. The Tree Replaced by Parallel Processors

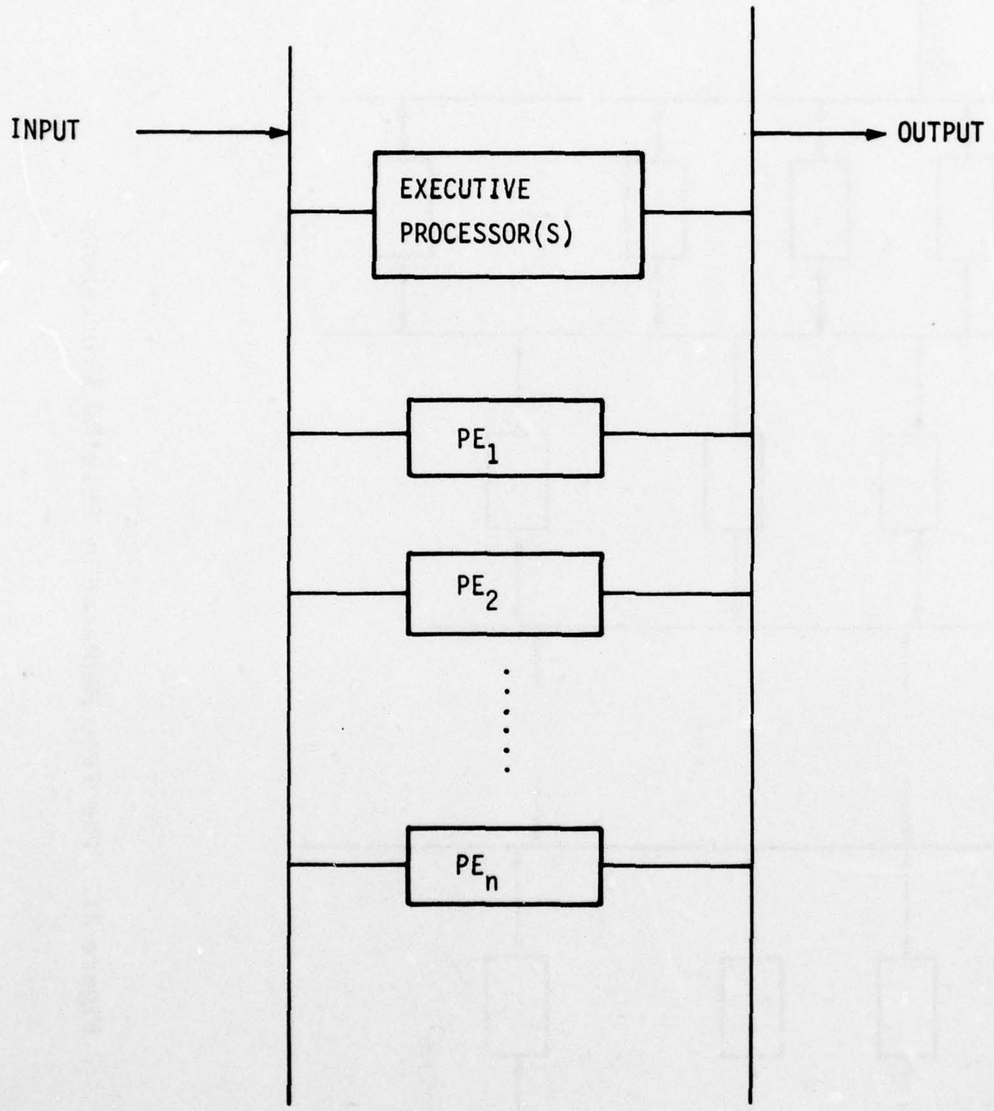


Figure 32. Designer D's Final Architecture
(All PE's have identical hardware.)

In Step 8, the designer specifies the system control algorithms to be used in the system. A single processing element is chosen to perform executive functions. Standard redundancy techniques are used to detect executive failures. A second processing element which has been parroting the executive functions always stands by to assume the control function whenever a failure is detected. When the backup processor begins serving as the executive, its first act is to appoint another good processing element to assume the parroting role. It must be initialized with the executive's state before normal processing can resume. The executive retains the following information:

1. Input files
2. List of available processing elements
3. Current radar status
4. Current task assignments
5. List of processing element status

This information is used to perform the following executive functions:

1. Transaction control (with associated bookkeeping)
2. Request distribution to processing elements
3. Broadcast current radar status to all processing elements

The design loop (Steps 7 through 11) must be traversed three times since there are three subfunctions. The second time around the designer included the request sorting subfunction. He observed that the logical tree structure is well suited to serial sorting; no redesign is required because the time not used for conflict processing can be used for request sorting. No major executive modifications are required to handle the new load; it must be able to schedule the new function on the processing elements, which is similar to scheduling constraint processing.

Finally, the schedule construction function must be included in the system. The designer hoped to use a tree-searching algorithm to perform schedule construction from the constraint schedule. Whenever only one request fits within a time interval, that request will be scheduled there. If several requests may be satisfied, priority-based selection is used. If this cannot resolve the conflict, the most attractive alternatives will be pursued by tree searching: separate processors are assigned different options, which then can be explored in parallel. The designer noted that the number of processors required to perform constraint processing and request sorting declines as the processing converges in the tree structure. Figure 33 illustrates this trend. Because the total processing capacity is constant, an increasing amount of resources can be devoted to schedule construction; this allows more alternatives to be explored as the schedule length increases.

3.4 OBSERVATIONS

We have four new architectures for the radar scheduling problem, but since they solved different variants of the problem and make different assumptions regarding element complexity and capability, we cannot directly compare their performance or other desirable attributes either among themselves or with extant architectures. It does seem, however, that conventional architectures do not deliver the required performance because they are limited to a single instruction stream.

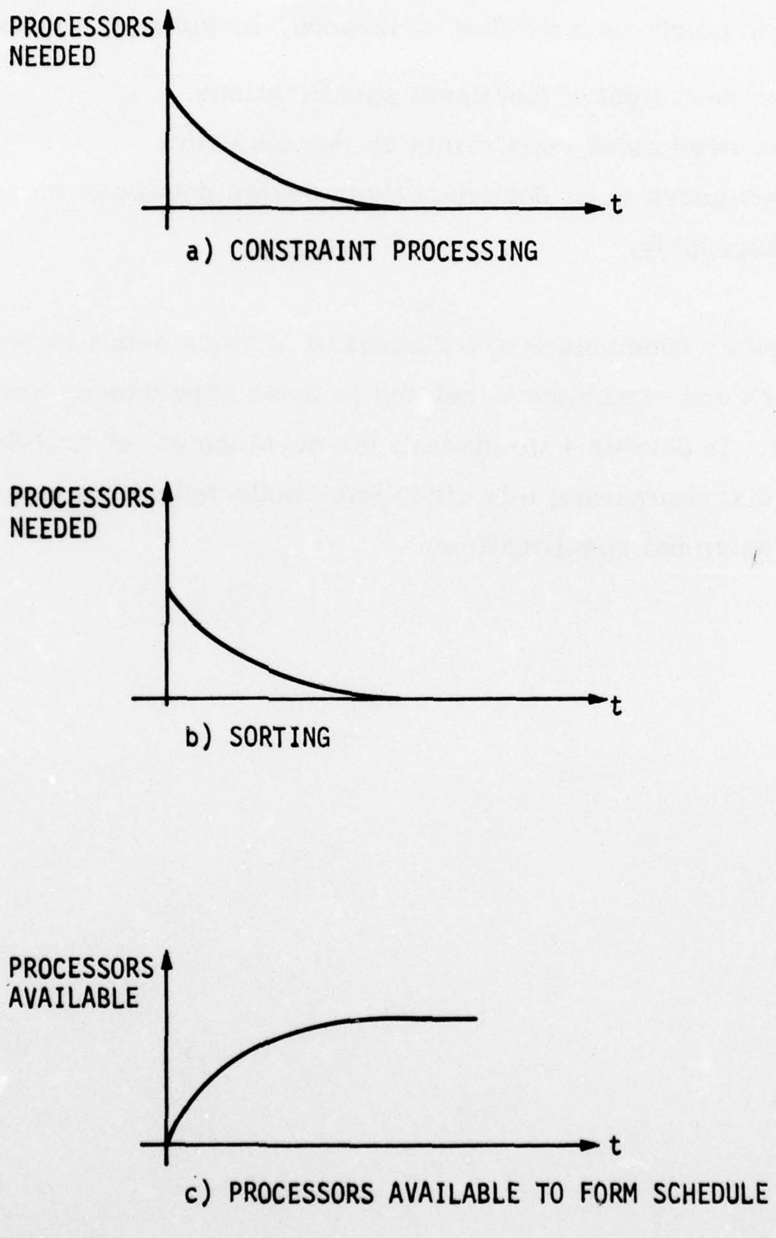


Figure 33. Processing Demands during Scheduling

This exercise taught us a number of lessons, including the following:

- We need tighter functional specifications.
- We need more constraints on the designers
- Designers must document their design decisions more thoroughly.

These and other conclusions are discussed in more detail in Section 6. Further work and experiments related to these experiences are discussed in Section 7. In Section 4 we discuss the development of architectures for optical discrimination; this study was conducted with much more control of functional specifications.

SECTION 4

OPTICAL DISCRIMINATION

Optical discrimination was the second BMD function studied in developing specialized distributed data processing architectures. This study was undertaken after the radar scheduling study was completed, which enabled us to incorporate some of the lessons learned from the radar scheduling experience into this study.

As we noted in Section 3.4, one major lesson from the radar scheduling exercise was the importance of giving the designers an exact functional specification. We attempted to attain this objective by having a meeting of the designers with those familiar with the application. The specification developed is described in Section 4.1, with realistic parameters discussed in Section 4.2.

Two designers independently developed architectures based on the common algorithm specifications. Sections 4.3 and 4.4 describe their approaches and the architectures they produced. A summary which includes comparisons of the architectures and the lessons learned during this design exercise is presented in Section 4.5. This design effort also produced numerous suggestions in experiments; these are discussed in Volume VII of this report.

AD-A047 475 HONEYWELL INC MINNEAPOLIS MINN SYSTEMS AND RESEARCH --ETC F/G 9/2
DISTRIBUTED DATA PROCESSING TECHNOLOGY. VOLUME IV. APPLICATION --ETC(U)
SEP 77 M G GOUDA , Y W HAN, E D JENSEN DASG60-76-C-0087
UNCLASSIFIED 77SRC68 NL

2 OF 2
AD
A047475

END
DATE
FILMED
1 -78
DDC

4.1 THE DISCRIMINATION FUNCTION

A discrimination function is a function associated with some sensor. The sensor scans its field of view periodically and reports on all objects found. The discrimination function examines all the data of those objects to decide which of them are real reentry vehicles and which of them are decoys. This decision is not binary. Actually, the discrimination function computes the value of the object ranking r for each object in the field of view; the larger r is, the more probable that the object is a reentry vehicle.

In this section, we specify the discrimination function used for our second design experiment. This function is strongly coupled with another function called the track function. Therefore, we specify here both functions and their interface--a data base called the track file.

First, the operations performed by both functions and the interactions between them are outlined. Then detailed specifications of the two functions are discussed.

4.1.1 An Outline of Some Track and Discrimination Functions

Figure 34 shows an arrangement of two files and three functions. The input data arrive as a sequence of "frames," where each frame contains data about all objects observed in one scan of the sensor's field of view. A frame consists of a number of records; each of them holds the data associated with one observed object, as shown in Figure 35. The data of one observed object are (t, p, $\underline{c}$) where t is the observation time,

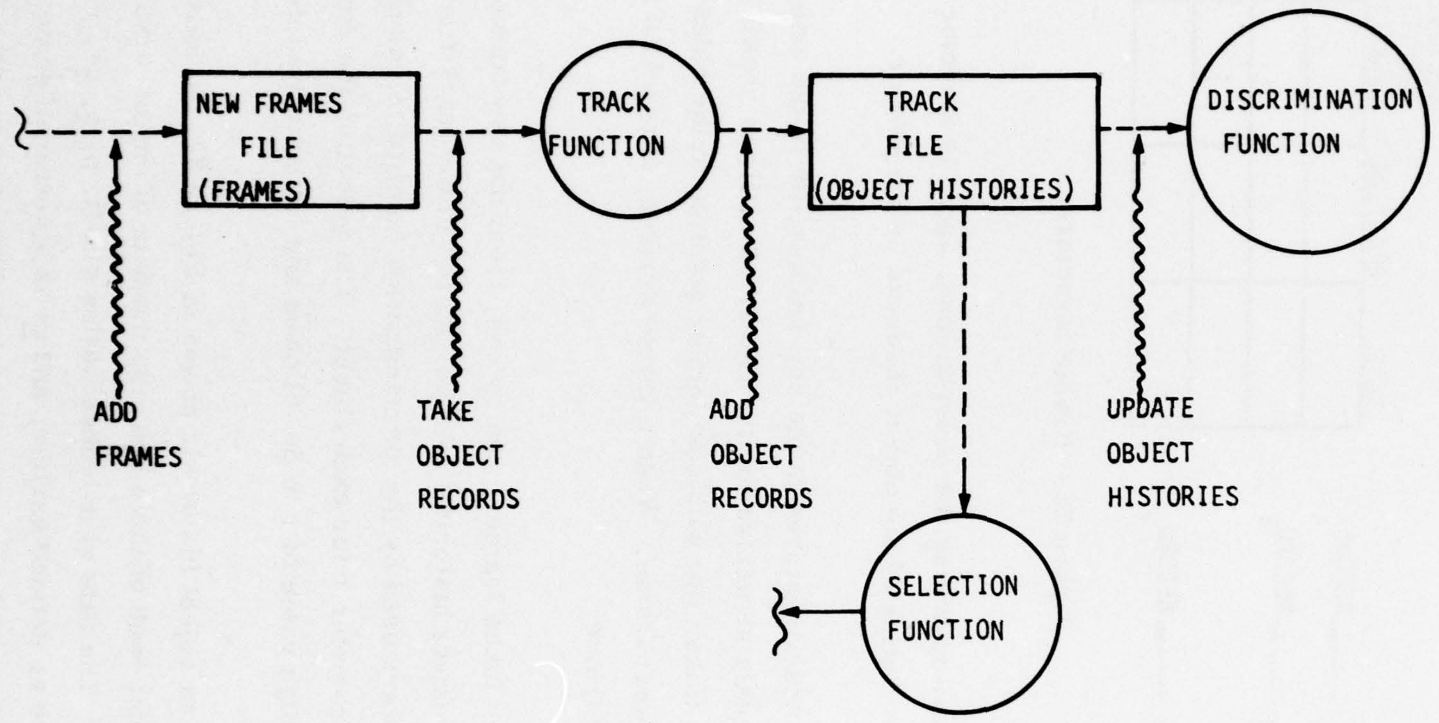


Figure 34. Arrangement of Some Track and Discrimination Functions

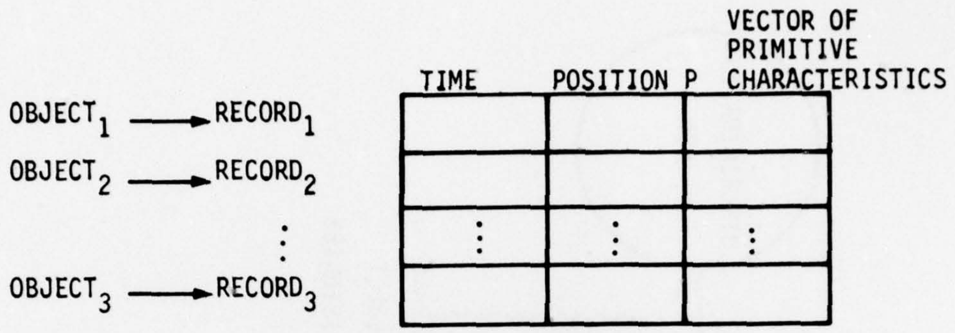


Figure 35. Frame Structure

p is the object position during the observations, and $\underline{c}$ is a vector of primitive characteristics of the object observed by the sensor.

The received frames are stored in the new frames file in the order of their arrival. A data structure for the new frames file is shown in Figure 36; it is a linked list with one pointer pointing to the oldest frame and one to the newest frame. When a frame arrives, it is added to the list as its newest frame.

The track function takes frames, one by one, from the new frames file and updates the "object histories" already stored in the track file. These updated histories are used by the discrimination function to compute new values for the parameter r for each object. The selection function selects an object with a high value of r to be followed and (ultimately) intercepted.

The structure of an object history is shown in Figure 37. It consists of a number of records; each of them contains the data of one observation for the same object. The data of one observation are (t, p, $\underline{c}$, $\underline{c}^*$, r) where t, p, $\underline{c}$, and r are as defined earlier, and $\underline{c}^*$ is a vector of some high-level characteristics computed from the primitive characteristics $\underline{c}$. Both $\underline{r}$

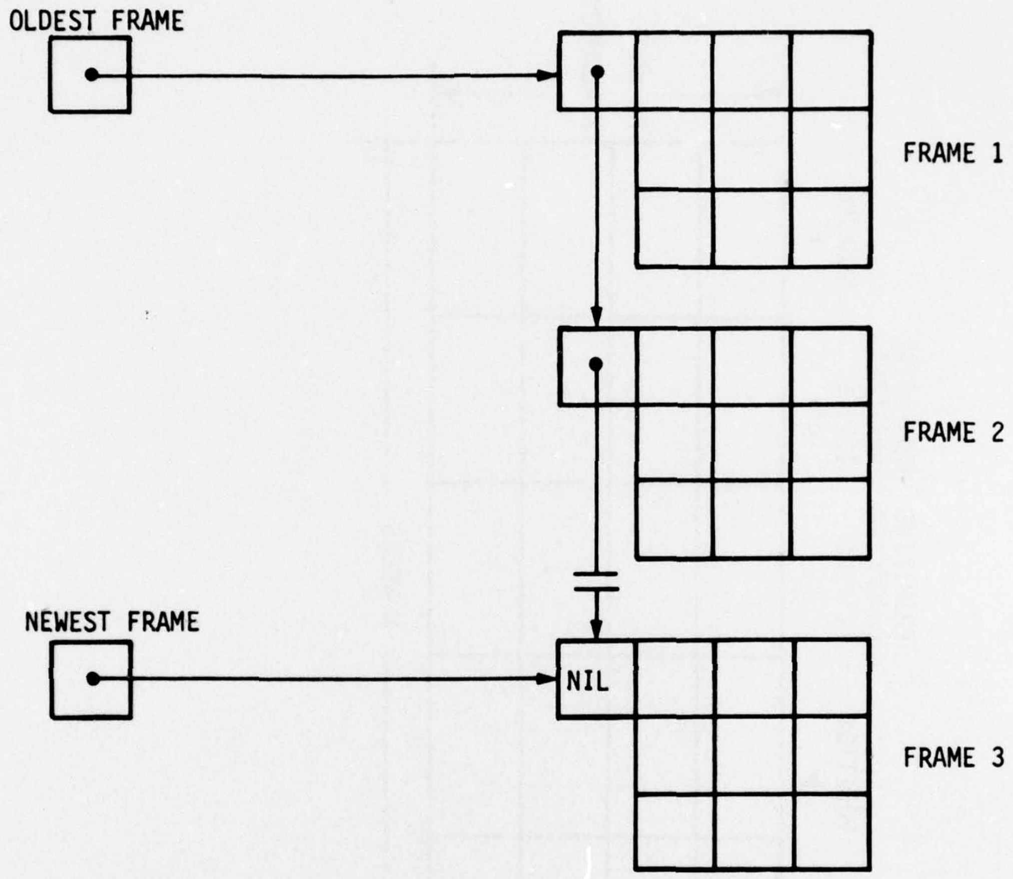


Figure 36. Data Structure for the New Frames File

and $\underline{c}^*$ are computed by the discrimination function, as explained in detail in Section 4.1.3.

The size of the track file is determined by two parameters M and H (Figure 37) where M is the number of words in each record in any object history, and H is the maximum number of records in any object history. (At any time instant, each object history contains, at most, the H most recent records of the object's history.) The maximum size of track file is MH words.

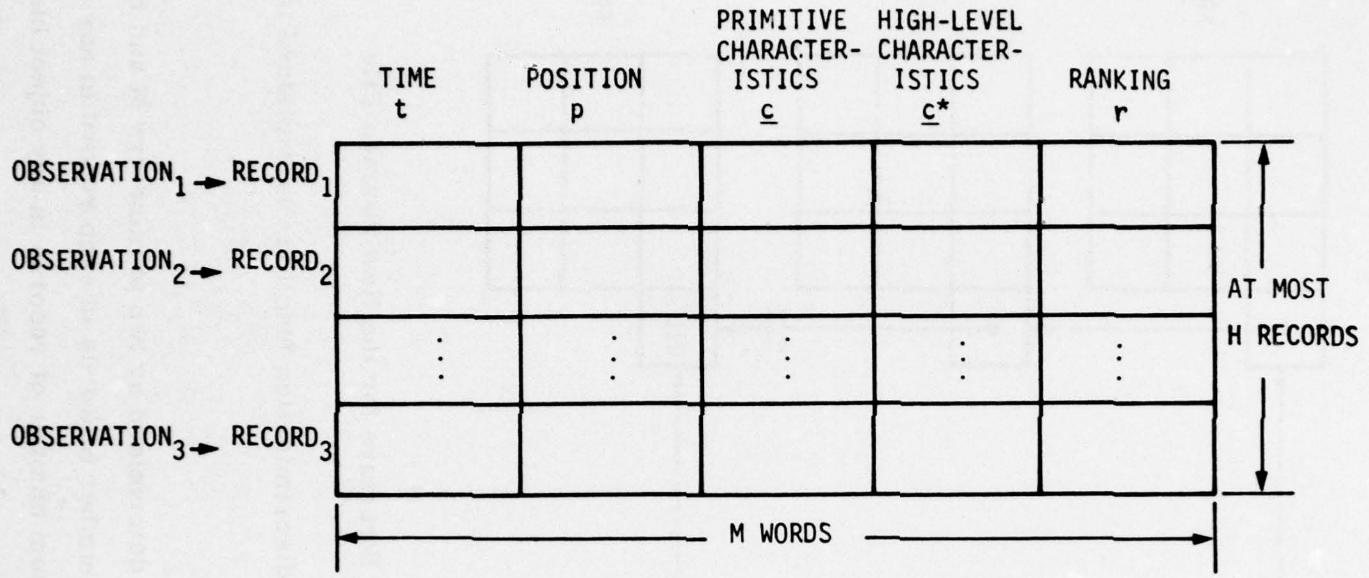

TIME t
POSITION p
PRIMITIVE CHARACTER-ISTICS c
HIGH-LEVEL CHARACTER-ISTICS c*
RANKING r
OBSERVATION$_1$ → RECORD$_1$
OBSERVATION$_2$ → RECORD$_2$
OBSERVATION$_3$ → RECORD$_3$
AT MOST H RECORDS
M WORDS

Figure 37. Object History Structure

The track function takes the frame records (t, p, $\underline{c}$) from the new frames file and correlates each of them with the object histories in the track file. (A data structure for the track file is shown in Figure 38.) The discrimination function computes the values of $\underline{c}^*$ and r for all added records. The track and discrimination functions are discussed in more detail in the next two sections.

4.1.2 A Track Function

Figure 39 shows the flowchart for a track function. The function operates on the frames in the new frames file, one by one, starting with the oldest frame. The operation stops when there are no more frames in the new frames file, and it resumes automatically on the arrival of more frames to the new frames file.

To operate on a frame, the function takes each frame record and either adds it to one of the object histories already stored in the track file or adds it to the track file as the first record in the history of a new object.

To decide whether a record (t_o, p_o, $\underline{c}_o$) belongs to a new object or to an old object, the track function compares its position p_o and its primitive characteristic $\underline{c}_o$ with estimates of the expected position p_i and characteristic $\underline{c}_i$ of all the objects in the track file at time t_o. If there is an object whose expected position p_i and characteristics $\underline{c}_i$ at time t_o are within small bounds from p_o and $\underline{c}_o$, respectively, then the record (t_o, p_o, $\underline{c}_o$) is added to the history of this object. In case there is no such object, the

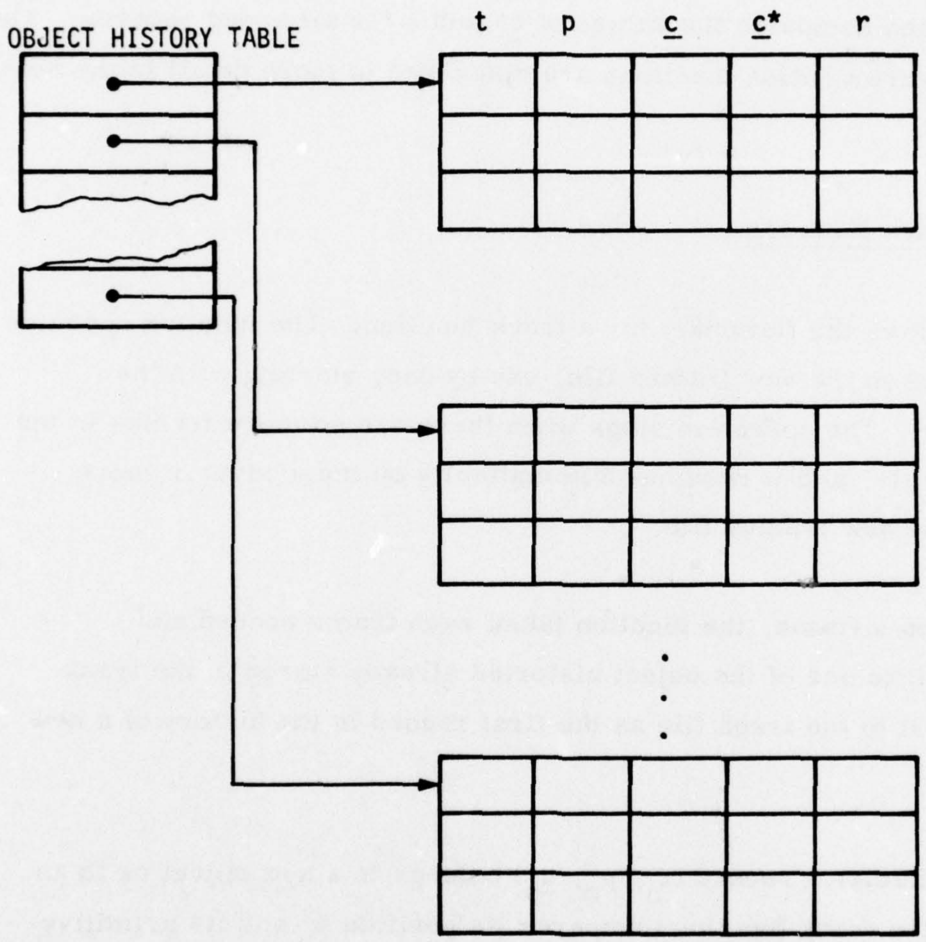


Figure 38. Data Structure for the Track File

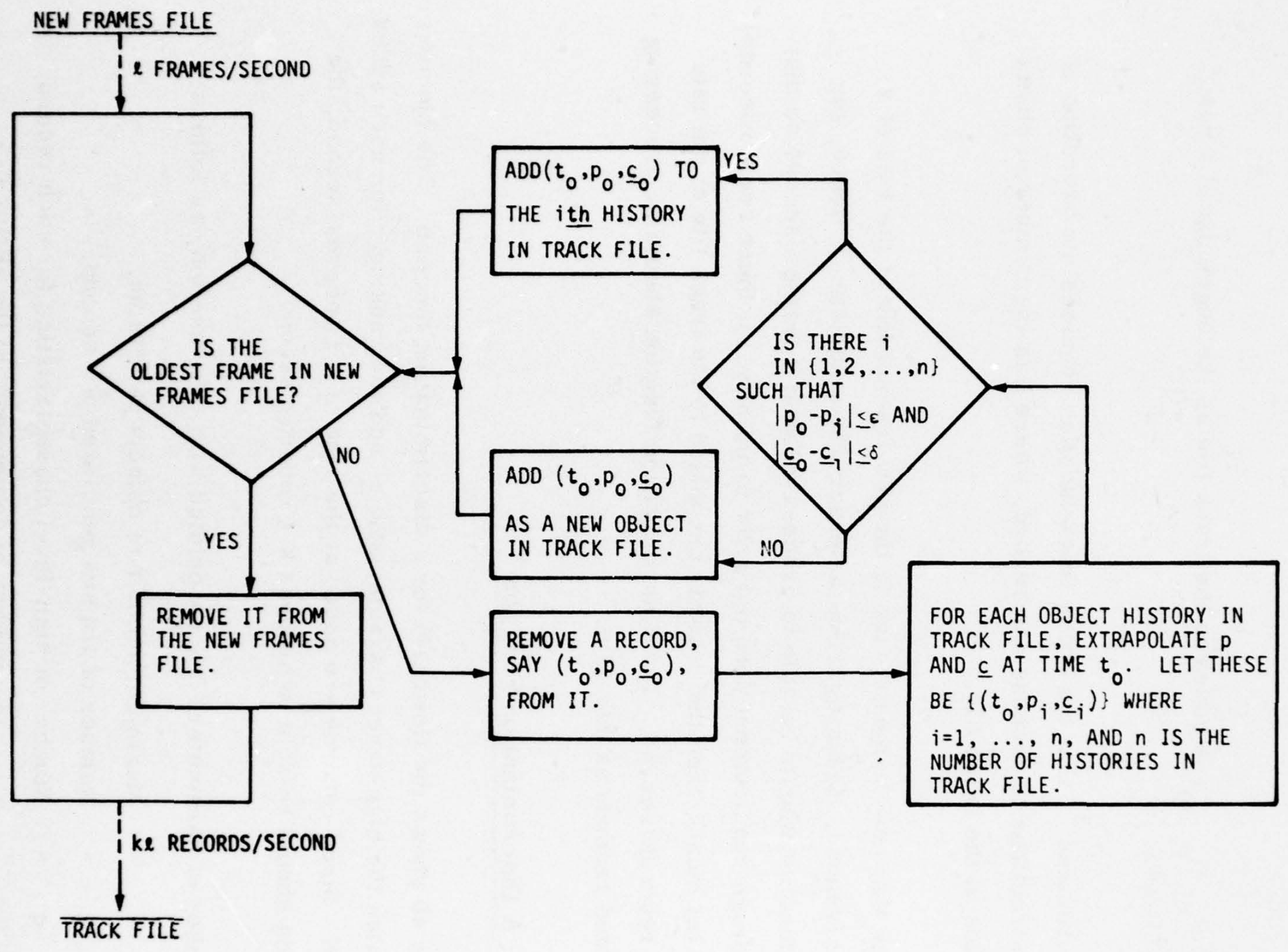

NEW FRAMES FILE
ℓ FRAMES/SECOND
IS THE OLDEST FRAME IN NEW FRAMES FILE?
YES
NO
REMOVE IT FROM THE NEW FRAMES FILE.
REMOVE A RECORD, SAY $(t_o, p_o, \underline{c}_o)$, FROM IT.
FOR EACH OBJECT HISTORY IN TRACK FILE, EXTRAPOLATE p AND $\underline{c}$ AT TIME t_o. LET THESE BE $\{(t_o, p_i, \underline{c}_i)\}$ WHERE i=1, ..., n, AND n IS THE NUMBER OF HISTORIES IN TRACK FILE.
IS THERE i IN {1,2,...,n} SUCH THAT $|p_o - p_i| \le \varepsilon$ AND $|\underline{c}_o - \underline{c}_i| \le \delta$
YES
NO
ADD $(t_o, p_o, \underline{c}_o)$ TO THE ith HISTORY IN TRACK FILE.
ADD $(t_o, p_o, \underline{c}_o)$ AS A NEW OBJECT IN TRACK FILE.
kℓ RECORDS/SECOND
TRACK FILE

Figure 39. Flowchart for a Track Function

record (t_o, p_o, $\underline{c}_o$) is added to the track file as the beginning of a new object history.

It is estimated[35] that the above track function requires the execution of 36h instructions to process one record, where h is the number of object histories in the track file.

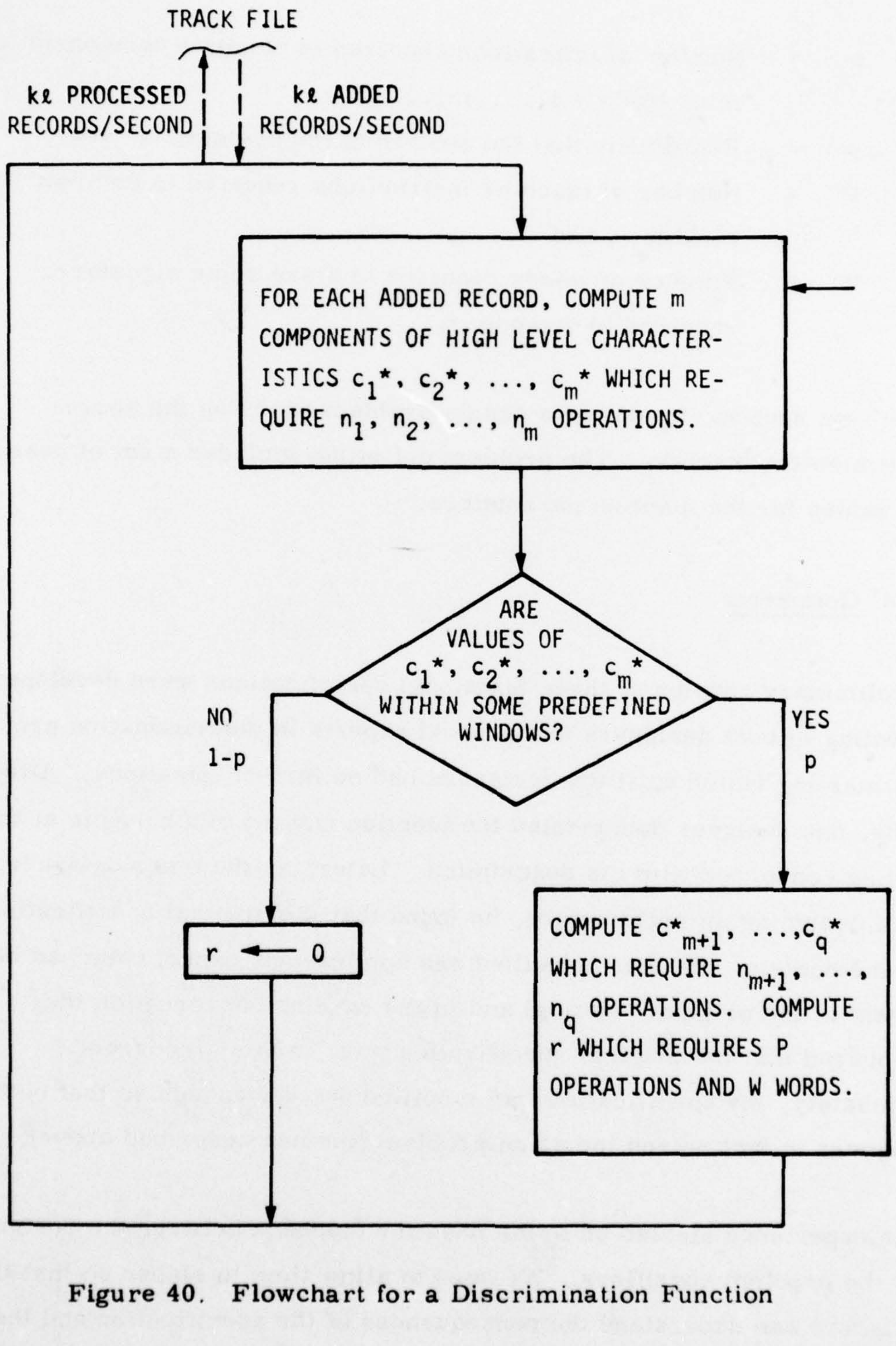
Assume that new frames arrive to the new frames file at the rate of ℓ frames/second. (This figure is a sensor characteristic.) Hence, the track function should be able to process at least ℓ frames/second so that the storage requirement does not grow to infinity. If there are k records/frame (at most), then the records are added to the track file at the rate of k ℓ records/second. The discrimination function should be processing the added records at this rate.

4.1.3 A Discrimination Function

Figure 40 shows the flowchart for a discrimination function. The function computes the high-level characteristic $\underline{c}^*$ and the ranking r for each added record. Since records are added at the rate of k ℓ records/second, the function should be able to process k ℓ records/second.

A number of parameters are associated with this function, as follows:

k	=	Maximum number of objects per frame,
ℓ	=	Number of frames generated per second,
q	=	Number of high-level characteristics for each record, i.e., the number of components in the vector $\underline{c}^*$,



TRACK FILE
kℓ PROCESSED RECORDS/SECOND
kℓ ADDED RECORDS/SECOND
FOR EACH ADDED RECORD, COMPUTE m COMPONENTS OF HIGH LEVEL CHARACTERISTICS c_1^*, c_2^*, ..., c_m^* WHICH REQUIRE n_1, n_2, ..., n_m OPERATIONS.
ARE VALUES OF c_1^*, c_2^*, ..., c_m^* WITHIN SOME PREDEFINED WINDOWS?
NO
1-p
YES
p
$r \leftarrow 0$
COMPUTE c^*_{m+1}, ..., c_q^* WHICH REQUIRE n_{m+1}, ..., n_q OPERATIONS. COMPUTE r WHICH REQUIRES P OPERATIONS AND W WORDS.

Figure 40. Flowchart for a Discrimination Function

n_i	=	Number of operations required to compute compoment $\underline{c}_i^*$ of $\underline{c}^*$ (for $i = 1, \ldots, m, \ldots, q$).
p	=	Probability that the answer to the decision is "yes,"
P	=	Number of machine instructions required to compute the ranking r, and
W	=	Number of words required to store some signatures required to compute r.

In the next section, we define a design problem based on the above discrimination function. The problem definition includes a set of reasonable values for the function parameters.

4.1.4 Comments

A preliminary version of these functional specifications were developed at a meeting of both designers and several experts in discrimination problems. That meeting lasted until the designers had no further questions. Afterwards, one designer documented the function and the other people at the meeting concurred with his description. Later, as the other designer began designing an architecture, he found that the original specification was incomplete. He then consulted one applications expert (who had been present at the original meeting) and in the ensuing conversation they discovered that the original specification was, in fact, incorrect. Fortunately, the specification was modified quickly enough so that both designers in fact solved the same problem (the one described above).

This experience alerted us to the need for feedback between the designers and the problem specifiers. We need to allow time to elapse so that the designers can understand the consequences of the specification and then ask clarifying questions. A single meeting is not adequate for this purpose. Other relevant points are covered in Section 6.2.1.

4.2 A COMPUTER ARCHITECTURE DESIGN PROBLEM

In this section, we define a problem of designing a distributed computer architecture to accommodate both the track file and the discrimination function defined in Section 4.1. The problem definition consists of assigning some numerical values to the file and the function parameters. A reasonable set of values for these parameters is as follows:

(1) Track File:

In the worst condition, the track file contains 50n object histories (where n ranges from 1 to 100). Each history contains 100 records; each record has 30 words.

(2) Discrimination Function:

k	=	50n records/frame (where n ranges from 1 to 100)
ℓ	=	2 frames/second
q	=	10 high-level characteristics (in vector $\underline{c}^*$)
m	=	8 high-level characteristics
n_1	=	$n_2 = \ldots = n_9$ = 80 machine instructions
n_{10}	=	300 machine instructions
p	=	.8
P	=	1000 machine instructions
W	=	100 words

The number of objects per frame (k) depends on the number of enemy threat clouds (n) against which the discrimination function is working. Each threat cloud may contain up to 1000 objects; most of these are decoys.

However, it is estimated[29] that, by the time the data reach the new frames file (after being filtered to decrease the number of observed objects in the sensor field of view), each threat cloud contributes around 50 objects to the frame. Hence, k equals 50n. In most cases, n ranges from 1 to 100.

We have chosen the parameter n to reflect the growth requirement of the discrimination function. This modifies the problem statement to become as follows: it is required that a family of distributed architectures be generated to accommodate the discrimination function and the track file such that each family member corresponds to at least one value of n, where n ranges from 1 to 100.

This problem was given to two designers who independently developed architectures based on the above specification. Their approaches and the architectures they produced are described in Sections 4.3 and 4.4. A comparison of their architectures and the lessons learned during the design exercise are presented in Section 4.5.

4.3 DESIGNER E'S ARCHITECTURE

This section describes the efforts of Designer E in developing an architecture for the discrimination function. In Section 4.3.1 we present the development sequence used the this designer. No methodology (in the sense of Section 2) was followed. In Section 4.3.2 we discuss the final architecture and a verbal description of how functions are mapped into the architecture. Finally, we present comments and observations concerning this effort and its outcome.

4.3.1 Architectural Development Process

Designer E produced one family of distributed architectures to perform the descrimination algorithm. In this section, we describe how he proceeded in his development of the design. The process will be explained in terms of a sequence of steps. Readers, however, should not associate this step sequence with a methodology (as described in Section 2) since the designer views these steps as specific to this particular function and the development of this particular architecture to perform the function.

In a preliminary study, this designer developed an architecture for the function and studied the imbalances in processing requirements. These considerations led him to contact the algorithm expert; it was determined that the specifications agreed to in the initial meeting were incorrect (see Section 4.3.3). Since these considerations did not directly influence the final architecture, we will not give their details. The remaining discussion in this section describes the development steps following the modification of the algorithm specification.

4.3.1.1 Step 1--Since the algorithm specification was very precise, the first attempt was to construct an architecture exactly parallel to the algorithm specification. Data from various objects would be pipeline-processed through the structure (shown in Figure 41). Next, the designer translated algorithm processing requirements specified in numbers of instructions executed per object to requirements on the modules. Table 2 summarizes these requirements. In addition, the number of data base entries referenced by each module had been provided in the specification. These are listed in Table 3.

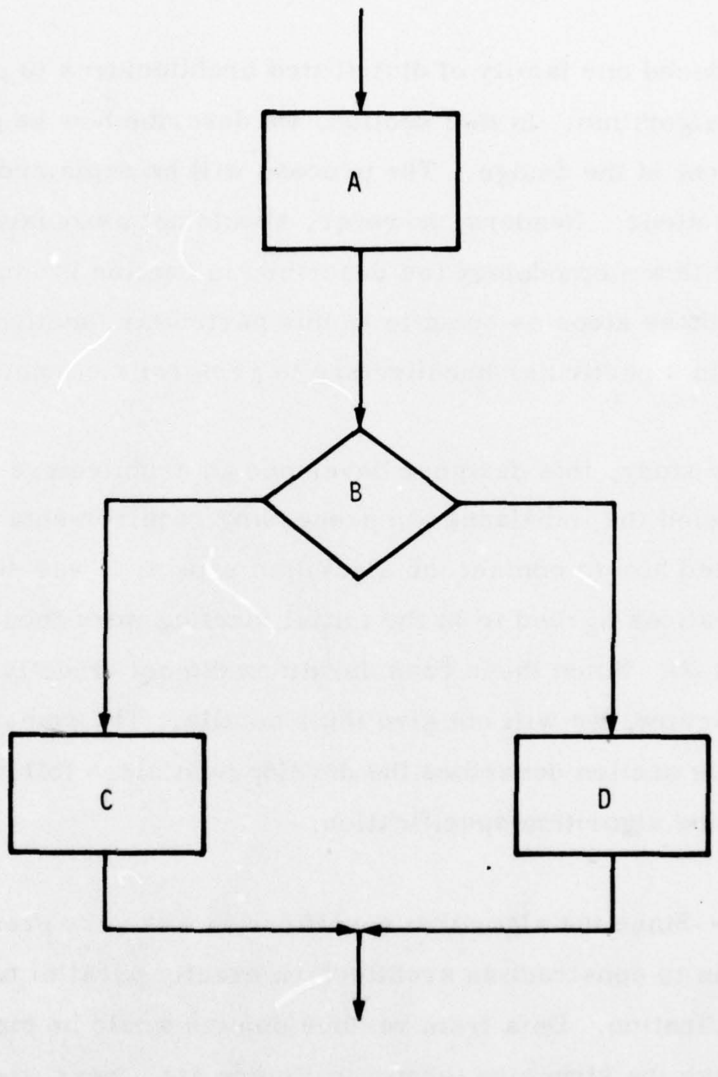


Figure 41. A First Pipelined Architecture

TABLE 2. PROCESSING REQUIREMENTS FOR THE ARCHITECTURE OF FIGURE 41

Module	Instructions/Object
A	650
B	40
C	10
D	1400

TABLE 3. DATA BASE ENTRIES REFERENCED BY MODULES IN THE ARCHITECTURE OF FIGURE 41

Module	Entries Referenced
A	0
B	16
C	0
D	100

There are several problems with the Figure 41 architecture. First, while the architecture shows how the processing requirements might be met, it does not show where the data will be stored. Second, pipelining will not work too well due to the decision element (box B) which causes different data items to be processed quite differently. Third, the processing requirements are unevenly distributed among the elements.

4.3.1.2 Step 2--Next, the designer attempted to alleviate the imbalance in processing requirements. He first observed that the C processing element would be invoked with low probability and that its only function was to set r to zero. Since D computes r when it should be nonzero, the C element can be removed if the A element zeroes r.

The designer noted that another approach to this problem is to send parameters to the D box chosen so that they result in an r value of zero. This is not a good design strategy, however, because any changes in the D algorithm require changes in the parameters. Furthermore, the D module must bear the brunt of the processing load even without processing the r=0 cases. Also, the setting of those parameters would be more complex than just clearing r to zero.

Note that the designer has not addressed memory issues.

4.3.1.3 Step 3--In this step the designer evaluated the processing requirements in terms of known processing elements. The parameter n, representing the number of threat clouds, is carried through this computation. A microcomputer capable of 100,000 operations/second was assumed. This instruction rate was deliberately selected to be somewhat lower than the raw processing rate of available machines to compensate for possible requirements on the precision of the computations. Note that the precision requirements were not specified with the algorithm specification. This oversight should be corrected in future specifications. Furthermore, the precision of the operations counted in the specification were not known.

The designer had to consider how to include the probabilistic nature of the decisions in estimating the computational requirements for the D element. For this preliminary design, the conservative approach was taken: the designer assumed that all objects would require D's processing. Table 4 shows the requirements expressed as the number of microcomputers necessary to meet the processing needs. These needs are determined without worrying about management overhead requirements.

TABLE 4. COUNTS OF MICROPROCESSORS NEEDED TO MEET PROCESSING REQUIREMENTS

Module	Microprocessor Count
A	0.65n
B	0.04n
D	1.4n

At this point, the designer still does not have memory explicitly shown. He also has not considered how to interconnect and control the processors.

4.3.1.4 Step 4--In the next step, the designer considered the interconnections required to communicate data and control information among the microprocessors. As a first step, he positioned a simple bus between the sets of processors in each module, as depicted in Figure 42. In addition to the problem that there is a central element that may be a bottleneck or a failure point, there seemed to be a problem with controlling such a large number of cooperating processors.

The designer thought that the control problem, growth requirements, and reliability requirements would be better met by a set of clusters of processors. Figure 43 illustrates this structure. Here each cluster maintains all information about an assigned set of objects. The track function assigns each observation in the track file to a single processor cluster.

A technique to improve the reliability by adding extra communications paths was explored. Figure 44 shows this structure. Additional bus coupling modules connect each bus to its (cyclical) neighbors in the structure. Data and control information are passed along these paths if elements fail beyond the recovery capability within the cluster.

Cluster sizes and memory arrangements were not studied in this step.

4.3.1.5 <u>Step 5</u>--The designer considered the maximum configuration of the system and estimated the maximum redundancy needed to meet reliability requirements. These studies were used to estimate the potential number of modules and clusters required. No effort was made to estimate the traffic among clusters. Very rough reliability estimates showed that 10 percent additional processing elements would be more than adequate to meed reliability needs; however, when less than 10 elements are required to meed functional needs, two additional elements are more than adequate.

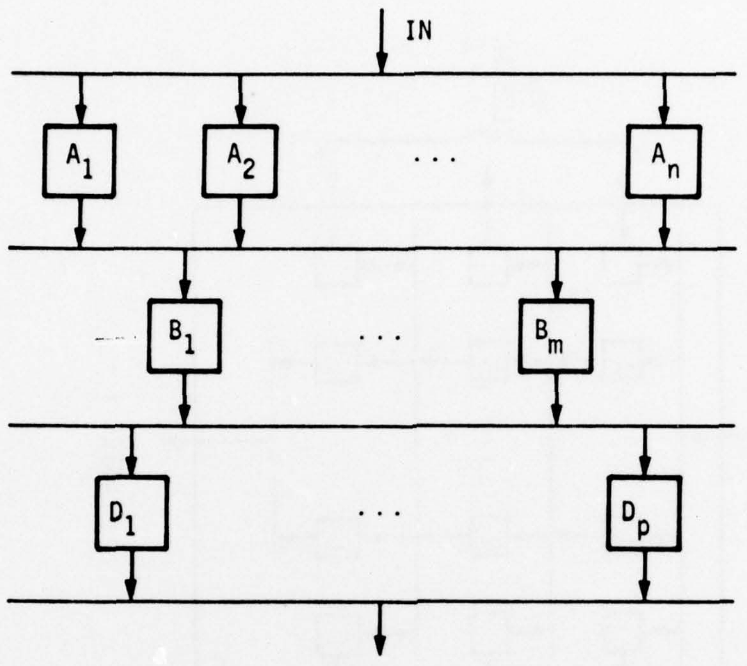


Figure 42. Processor Sets To Implement the Modules of Figure 41

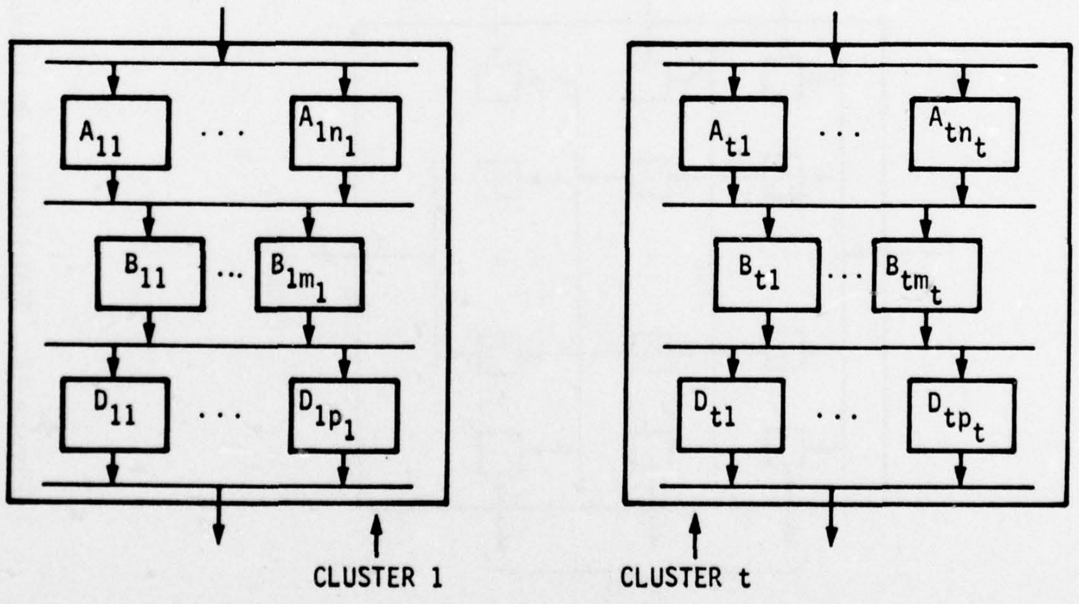


Figure 43. Processor Clusters

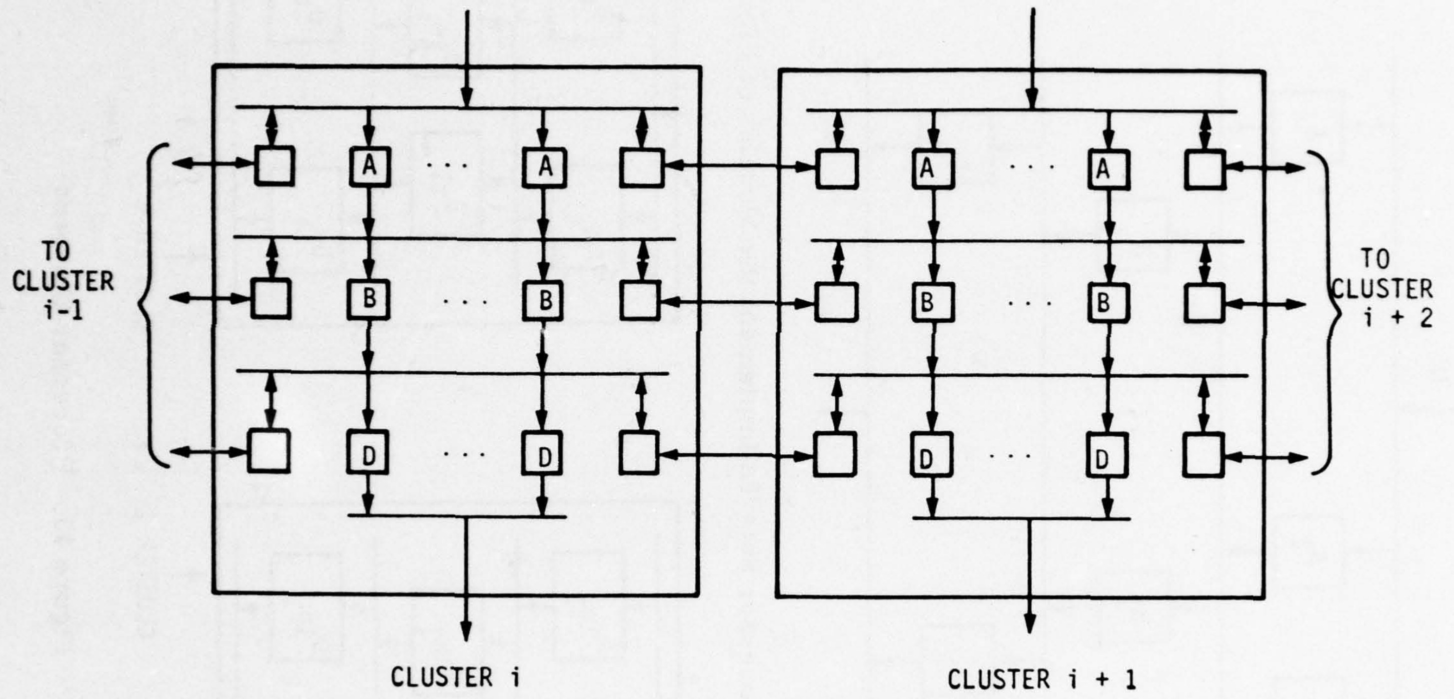


Figure 44. Multicluster Architecture with Intercluster Coupling

These calculations assumed that the individual elements were reliable and that the system was to be ultra-reliable (with failure probability two orders of magnitude lower).

On the basis of these calculations, the designer decided that the system could use 8-bit addresses to select modules within classes.

4.3.1.6 Step 6--The designer reconsidered the modularization of the computations. Since the B elements performed so little processing, their functions were combined with the A elements to produce the revised cluster architecture shown in Figure 45. Memory requirements are still missing.

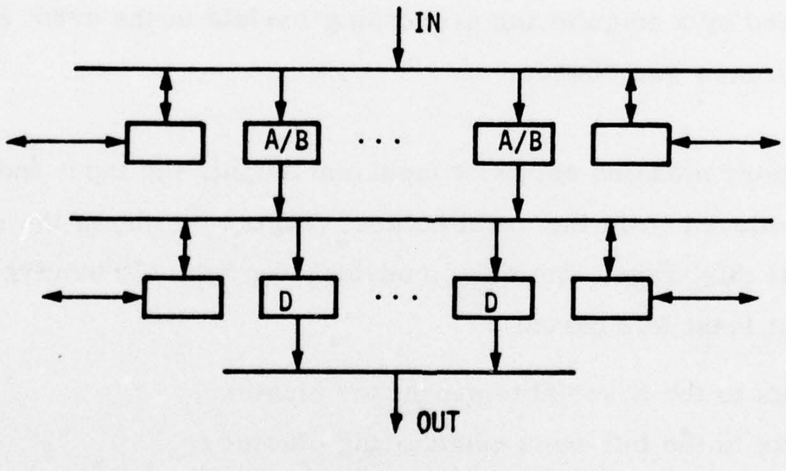


Figure 45. Revised Cluster Architecture

4.3.1.7 Step 7--The designer, being reasonably content with the cluster structure with respect to processing requirements, turned to memory requirements. Two observations are relevant. First, memory is actually the input and output since the discrimination function actually examines new entries in the track file and produces evaluations within the track file. Second, the memory could be used for intercluster communications.

Two approaches were considered. Each memory module could have a single port and could be connected through a bus to all elements requiring access to the module. Alternately, the modules could have many ports connected to different buses, each associated with the cluster that requires access to the module. The former approach results in a single bus connecting every track file memory to every module since each track file will be accessed by a neighboring processing module in the event of failure. The latter approach was chosen.

Since the memory modules serve as input and output, the input and output buses were removed from the architecture. Figure 46 shows the cluster architecture at this stage. Memory modules, represented by circles, are connected to at least five buses:

1. Bus to the associated processor cluster
2. Bus to the left-hand neighboring cluster
3. Bus to the right-hand neighboring cluster
4. Bus from the track function
5. Bus to the function using the discrimination results

These connections are not all illustrated in Figure 46. Addressing issues associated with these connections are discussed under Step 8. The connections to neighbors are arranged in a circular ("ring") pattern to provide neighbors for the first and last clusters.

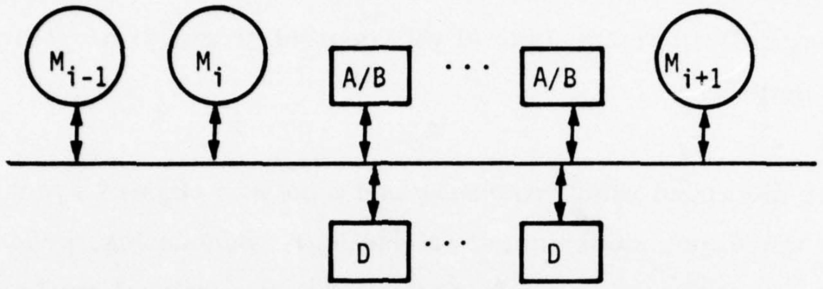


Figure 46. Final Cluster Architecture

The designer considered possible systems control algorithms for use with this structure as he selected the structure. Step 9 discusses these algorithms.

4.3.1.8 Step 8--The designer assigned memory address spaces to the memory modules. A uniform addressing scheme is required if common algorithms are to be used in each cluster. The two most significant address bits select among the three memory modules that may be accessed from any bus. A possible assignment of modules to these bits is specified in Table 5.

TABLE 5. MEMORY MODULE SELECTION ASSIGNMENT

Most Significant Address Bits	Module Selected
00	Within cluster controlling the bus
01	Within left-hand neighbor
10	Within right-hand neighbor
11	Reserved for additional connections

4.3.1.9 Step 9--The designer refined his notions of how the complete system would be controlled. As he developed the Step 7 architecture, he formed some preliminary designs of this control strategy; he returned to fill in more details.

The designer identified what processes and data will require synchronization (and, therefore, coordinated scheduling). Scheduling, processor status, memory status, and input arrivals require nonlocal accass and possible synchronization. Failures affect processor status; this information is collected in the cluster's memory and also passed from processors to their neighbors to facilitate their ability to pick up the load whenever failures occur. A scheme was developed to pass processor failure information around the system so that the "well" modules could reroute requests and perform all necessary processing. Memory failures were considered to be just like failures of the associated processors. This view is not accurate, especially as regards the memories used to communicate information among processors.

The selected algorithms will be discussed in Section 4.3.2.

4.3.1.10 Step 10--The designer wanted to determine the sizes of clusters and the number of clusters. He would have used reliability and growth requirements to perform this step, but no such quantified requirements were available. Therefore, he simply identified the issues involved in these decisions; his list included the following issues:

- How much bandwidth will be required for intercluster communications after failures?

- How much memory redundancy is required? What if data are lost?
- May the system grow by pieces of clusters? If not, the cluster size might be decreased. If so, the clusters can be larger, which reduces the total amount of hardware required to achieve a certain reliability specification while keeping the intercluster bandwidth constant.

The cluster architecture shown in Figure 46 with the clusters interconnected in a ring was the final architecture of this designer's effort.

4.3.2 System Architecture and Operation

One problem in existing system description techniques is the difficulty in specifying how the functional algorithm and system control features are mapped into the hardware modules. Since we have not developed a systematic solution to this problem, we present this information in textual form. We divide the description according to the type of information. The first subsection describes how the applications data are stored in the memories and how they are routed there. Their processing is described in the second subsection. Reliability issues, including failure detection and reconfiguration, are covered in the third subsection. Finally, growth is discussed in the fourth subsection.

4.3.2.1 Data Storage--Track file formats were defined with the algorithm specification (see Section 4.1). Although the designer realized that these formats were flexible, he saw no need to change them except in distributing the information among the memory modules in the architecture. Each processing cluster holds track file information for a set of objects, along with the associated queue listing new records that have been added to the track file but not yet processed by the discrimination function.

Three issues will be discussed below:

1. Adding a new object to the track file,
2. Adding a new record for an object already in the track file, and
3. Failure modes.

4.3.2.1.1 New Objects--An object in the field of view is deemed to be a new object when its position does not match the predicted position of any previously known object. A new track file must be created. Since the track function relates to all objects, it must access all object histories in processing any new observation. This suggests that all track file memories should be in the same address space when they are accessed from the track function. We can achieve this effect by making all track file memories accessible on the same memory bus associated with the track function, as shown in Figure 47. There is no interleaving among the modules (internally they may be interleaved, but this lower level of detail is not covered here).

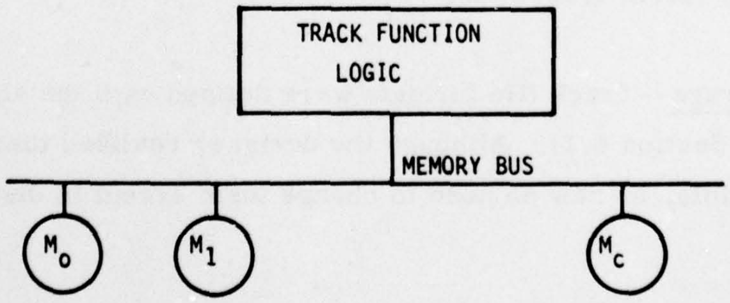


Figure 47. Memory Access to Track Files from the Track Function

Whenever a new object is detected, a track file is created in the next memory module selected on a round-robin basis (an alternate policy is discussed in Section 4.3.2.3). The track file algorithm must know how many memory modules exist to perform this function correctly. After the track file is created, a record is added to it using the same algorithm as for an old object.

4.3.2.1.2 New Records--Each track file is a circular buffer with associated control pointers. A new record is added to the file by incrementing the buffer input pointer (modulo the buffer size) and adding a pointer which describes the new record to the queue which describes records to be discriminated. Processors in the discrimination function examine this queue to find new work (see Section 4.3.2.2).

Each memory module contains a listing of all track files active within that module (this information could be replicated, if desired, to improve the system reliability). Thus, the data structure in each module resembles the overall track file format depicted in Figure 38.

4.3.2.1.3 Failure Modes--Two interesting approaches to memory failure are the following:

1. Assume that the entire associated cluster has failed. The track function would skip over that memory when assigning new tracks to modules. A list of functioning modules must be maintained within the track function.

2. Assume that the processors in the cluster are still operating correctly despite memory failure. In this case, place inputs for the failed memory into one of its neighbors.

Failure mode behavior is discussed in more detail in Section 4.3.2.3.

4.3.2.2 Processing--A/B processors that complete processing examine the queue of new track records to find new records to process. They must lock the queue to perform these operations. If the queue is empty, they place themselves on a special idle processor queue to await the appearance of new records in the track file. When the track processor inserts new records, it checks the idle processor queue to determine whether a processor is awaiting arrival of a new record. If a processor is waiting, it is informed of the existence of the new record and removed from the idle processor queue. The form of the notification has not been determined; possibilities include:

1. An interrupt signal, and
2. A special value in a memory location unique to the processor.

When an A/B processor has a record to process, it performs the first two elements of the processing as described in Figure 40. It evaluates eight characteristics of the object and determines whether these characteristics' values lie within the selection windows. In any case, it clears the value of r to zero. The A/B processors could be designed to process a set of records in pipeline fashion; the designer did not investigate this possibility in any detail since pipelining only improves performance, and he chose to get the required performance by processor replication. An interrupt signal is more attractive but requires additional hardware interconnections.

When an A/B processor completes processing, it decides whether the record requires D processing. Records requiring D processing are placed on a D scheduling queue; this is managed in the same manner as the A/B scheduling queue described above.

The D processor computes the two remaining characteristics and assigns a ranking to the object based upon the degree of match between its measured characteristics and the characteristics of known object classes. The final result of the processing is an r value stored in the track file.

The algorithm specification did not contain any information regarding further interfaces so the designer did not include any notification of the new r value to another system function. A queueing mechanism as described above can solve that problem, if necessary.

4.3.2.3 Failures--Failures introduce two needs: detection and reconfiguration. Many standard failure detection techniques could be used in this system; the designer did not study them in detail for this application. In addition, the processors can occasionally process synthetic data which should produce a known result. Idle periods are especially appropriate for such processing. The scheduling described in Section 4.3.2.2 could be altered as follows:

1. Each processor which finishes processing a record and finds the queue empty executes the test case.

2. If the test case succeeds, the processor again checks the request queue and places itself on the idle processor queue if there is still no work to do. A failure of the test case signifies a processor failure.

Another possibility is for the processors on the idle processor queue to execute diagnostic programs. Any failure would be communicated to a neighboring processor.

Whenever a failure is detected, the system must be reconfigured. This designer considered only reconfiguration by schedule modification. Failed processors are not scheduled for any record processing. Failed memories are prevented from receiving further unique information (information which, if lost, could not be found elsewhere in the system) by redirecting the processors producing the information.

The reconfiguration policy is to attempt to confine the reconfiguration to the cluster containing the failed element. A possible algorithm to implement this policy will be described below. If confinement fails, a reconfiguration with minimum effect on the neighbors is attempted. An entire cluster may fail, in which case it is removed from the track file assignment algorithm. We discuss these reconfiguration procedures below.

4.3.2.3.1 Local Reconfiguration--Local reconfiguration only covers processor failures. Memory failures were not extensively considered; redundant backup memories cause no reconfiguration visible to the software. If all members of the redundant set of memories fail, the cluster cannot accept records for processing in the normal way; therefore, this case can be considered a cluster failure.

When a processor failure is detected, it is removed from the scheduling function. At the same time, a signal is sent to the central control (mechanism not specified) to inform maintenance personnel of the failure. If the failure reduces the number of well processors in the cluster below a certain threshold, the target function is notified to reduce the number of new records assigned to this cluster. An alternate strategy is to assign records to clusters based on the numbers of well processors rather than the number of well clusters. This strategy could be accomplished by maintaining a list of well processors with the corresponding cluster numbers within the track function. Records would be assigned to clusters by cycling through the list of well processors, rather than by cycling through the set of clusters. Recall that the assignment of a record to a cluster does not imply an assignment of that record to the processor (which occurs later when processors become free). One problem with this strategy is that D processor failures are not adequately handled. More sophisticated scheduling/assignment policies were not studied by this designer.

Another strategy to overcome processor failures is for an idle processor to examine the schedule queues in both neighbors to find work when none is available locally. The processor would set itself idle only if no work were available locally or in the neighbors. This strategy is easily implemented because each cluster has access to the memories of its neighboring clusters. It will increase the intercluster communication bandwidth required.

These strategies do not consider graceful saturation when priorities associated with the records should be used to select those that do get processed. The function specification was missing any requirements in this area (see Section 6.2.1.11).

4.3.2.3.2 Cluster Failure--Cluster failures require more complex reconfiguration. The track function will not assign records to failed clusters by following a strategy such as those outlined above. Cluster failure complicates the redundancy since neighbors may be (logically) missing; the system could even be reduced to a set of nonconnected groups of clusters. Because many failures are required to create this situation, and since the system components are presumed to be reliable, system disintegration to this extent was not considered in this design exercise.

4.3.2.4 Growth--Threat growth is handled by adding new processors, memories, and even clusters to the system. Since the scheduling policies are table-driven, this growth affects only table sizes. All remaining issues are similar to the issues covered in the failure reconfiguration discussion in Section 4.3.2.3.

4.3.3 Observations

Designer E's observations are collected under four categories: reflections on the function specification, the emphasis in the architecture, the configuration parameters, and architectural description techniques.

4.3.3.1 Function Specifications--Designer E noted several omissions in the functional specification. First, the original specifications were incorrect and incomplete. It is difficult to verify correctness or completeness until a designer discovers that he needs more information.

Second, the specifications gave no information concerning degradation for saturation policies. These should be included in all functional specifications.

Third, the specifications did not clearly state whether the computations could be pipelined or whether significant data-dependent decisions were made within the algorithms.

Fourth, the specification should give a design parameter concerning data-dependent decisions. Though it was stated that the decision box (B) would produce one decision 80 percent of the time, it was not clear whether the design must efficiently handle higher loads. The designer proceeded conservatively in specifying a preliminary system configuration.

Fifth, the specification did not state how the output should be directed to succeeding modules. In this function, the output values are placed in a file, but some modules must require notification; this requirement was not stated.

Sixth, the specifications should state penalties for loss of information. The designer cannot guarantee perfect retention of information. Clearly he should be able to trade the retention probability against other design parameters. He was not given any information about the consequences of losing a track file on an object, for example. He could assume, for example, that loss of an object's track file is not important because future observations will create a new file. This would possibly delay identification of the object, but since there were no time deadlines imposed, the consequences of such a delay were unspecified. However, this loss may be unacceptable, in which case more memory redundancy will be needed.

4.3.3.2 Architectural Emphasis--In the absence of constraints on allowable module capabilities, the designer was forced to make reasonable choices. He then spent most of his effort on interconnection techniques, at both hardward and software levels. These techniques were oriented towards both static reconfiguration ("growth") and dynamic reconfiguration ("after failure"). Table-driven assignment/scheduling algorithms assist both needs. The absence of specifications on degraded systems performance made this design difficult.

The designer omitted interconnections necessary for central notification of module failures.

The designer observed that the interface between the track and discrimination functions has much architectural significance. If, for example, the track files are distributed to many memories for discrimination processing, the track matching can also be performed in parallel. Under this structure the interface is actually a parallel set of files. Architects of neighboring functions should consult closely about the interface structure so that both can reap all possible benefits from restructuring the interfaces.

During the course of the design, the designer identified several experiments to determine details of interconnection techniques. These suggestions are described in Volume VII of this report.

4.3.3.3 Configuration Parameters--Several configuration parameters were not studied either due to lack of sufficient data, lack of specifications, or lack of time. These parameters include:

- Effects of scheduling overhead on processor loads. In particular, are the execution sequence lengths compatible with the scheduling overhead?
- Cluster size. The cluster size affects growth patterns. Note that some failure policies permit normal operation with partially complete clusters (especially if the scheduling is based on active processors rather than active clusters); this makes cluster size more independent of growth problems. Cluster size also affects reliability, especially in the face of memory failure. The designer did not address these issues.

4.3.3.4 Description Techniques--The designer noted that it will be difficult to develop systematic descriptions of the architectures and systems control policies when they are incompletely developed. He did not see how to specify his partial design in a precise way except by using English text.

4.4 DESIGNER C'S ARCHITECTURE

This design is described in three parts. First, the functional requirements are determined; then architectures for discrimination and tracking are specified.

4.4.1 Functional Requirements for the Discrimination Function and the Track File

There are two types of functional requirements to be satisfied for each function: processing rate requirements and memory size requirements. In this section, we compute both requirements for the discrimination function. We also compute the memory size requirements for the track file.

4.4.1.1 Processing Rates for the Discrimination Function--Figure 48 shows the flowchart for the discrimination function; each box contains the number of machine instructions needed to execute the corresponding step (the steps are described in Figure 40). These numbers are computed from the parameter values given in Section 4.2. The average number of machine instructions needed to process one record is

$$\begin{aligned} & 640 + 80 + 0.2(2) + 0.8(1380) \\ = \ & 1824.4 \\ \approx \ & 2000 \quad \text{instructions/record} \end{aligned}$$

This is an average cost computation for a probabilistic cost machine.[30, 31]

Since records are added at the rate of 100 n records/second, the processing rate requirement for the discrimination function is

$$\begin{aligned} & 2000 \times 100\ n \\ + \ & 200\ n \quad \text{K instructions/second} \end{aligned}$$

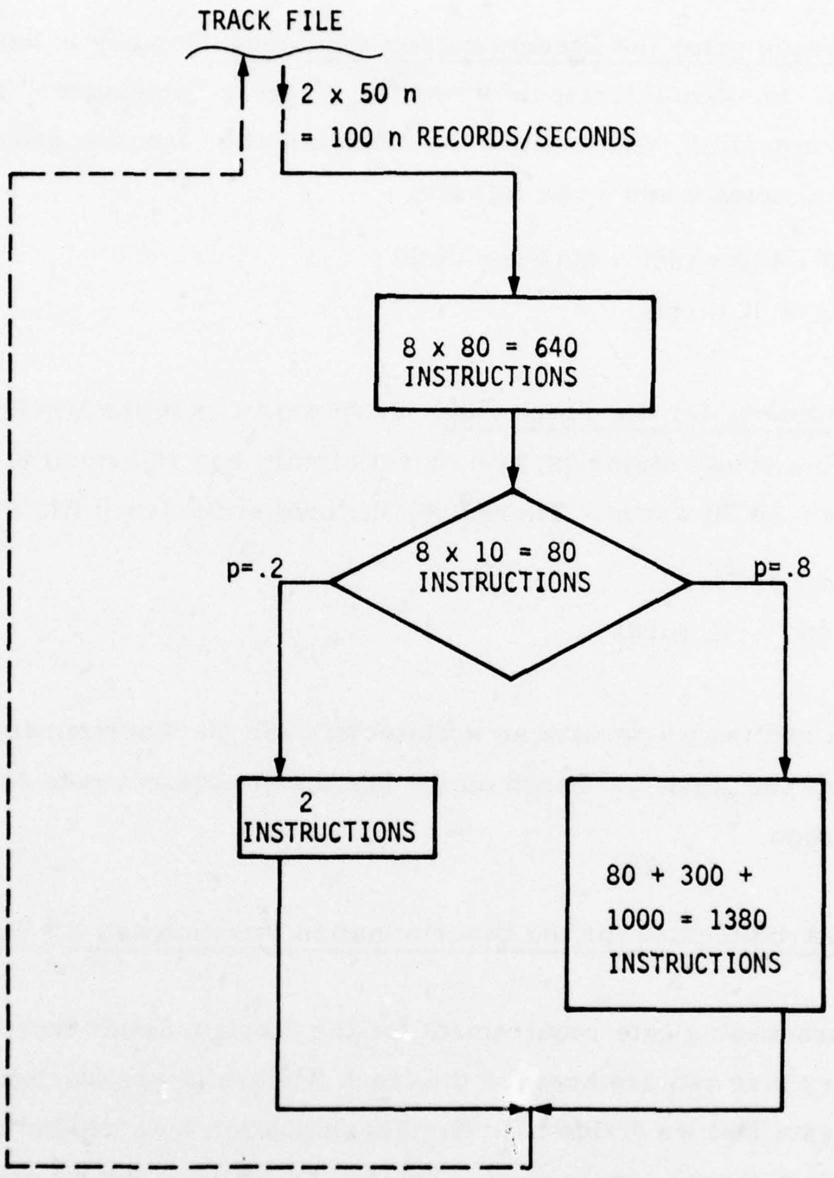


Figure 48. **Outline of the Flowchart of the Discrimination Function Showing the Number of Instructions Needed to be Executed at Each Step for Each Record**

4.4.1.2 <u>Memory for the Discrimination Function</u>--Memory is used in three ways: to store intermediate results, to store "signatures" for the ranking computation, and to store the discrimination function program. These requirements add up as follows:

$$100 + 100 + (640 + 80 + 2 + 1380)$$
$$\approx 2.5 \text{ K words}$$

4.4.1.3 <u>Memory for the Track File</u>--*In the worst* case the track file contains 50 n object histories. An object history has 100 records (at most); each of them as 30 words. Therefore, the size of the track file is

$$50n \times 100 \times 30$$
$$= 150\, n \text{ K words}$$

In the next section we present an architecture for the discrimination function and the track file based on the functional requirements computed in this section.

4.4.2 <u>An Architecture for the Discrimination Function and the Track File</u>

Both the processing rate requirement for the discrimination function and the memory size requirement for the track file are linear functions of n. This suggests that we divide both the discrimination function and the track file into n components each. This scheme allows for a linear (in n) architecture growth.

Figure 49 shows an architecture graph* based on the above scheme. It consists of n modules (recall n ranges from 1 to 100), each having one memory unit of 150 K words to accommodate one track file component, and one processing unit of (200 K instructions/second, 2.5 K words) to accommodate one discrimination function component.

A number of nonfunctional requirements are satisfied in the above architecture. They include protection, reliability (or graceful degradation under failure conditions), uniformity, and graceful growth.

4.4.2.1 Protection--Both the track file and the discrimination function are built using a number of memory and processing units. Each memory unit can be accessed by only one processing unit. Therefore, if one processing unit fails and continues to operate while its failure is undetected, the harm is limited to the memory unit connected to it. The rest of the track file remains unaffected.

4.4.2.2 Reliability--There are two types of possible failures for a processing unit. One type of failure causes the processing unit to stop operation. The other type of failure allows the processing unit to continue in its operation with some possible errors. Both types of failures can be handled in the above architectures.

If one processing unit halts, the track function processor is informed of its condition so that no more records will be added to its memory unit. These new records are directed to other memory units to begin building new object histories to replace the lost histories. For the track function

* Architecture graphs are defined in Section 2.1.

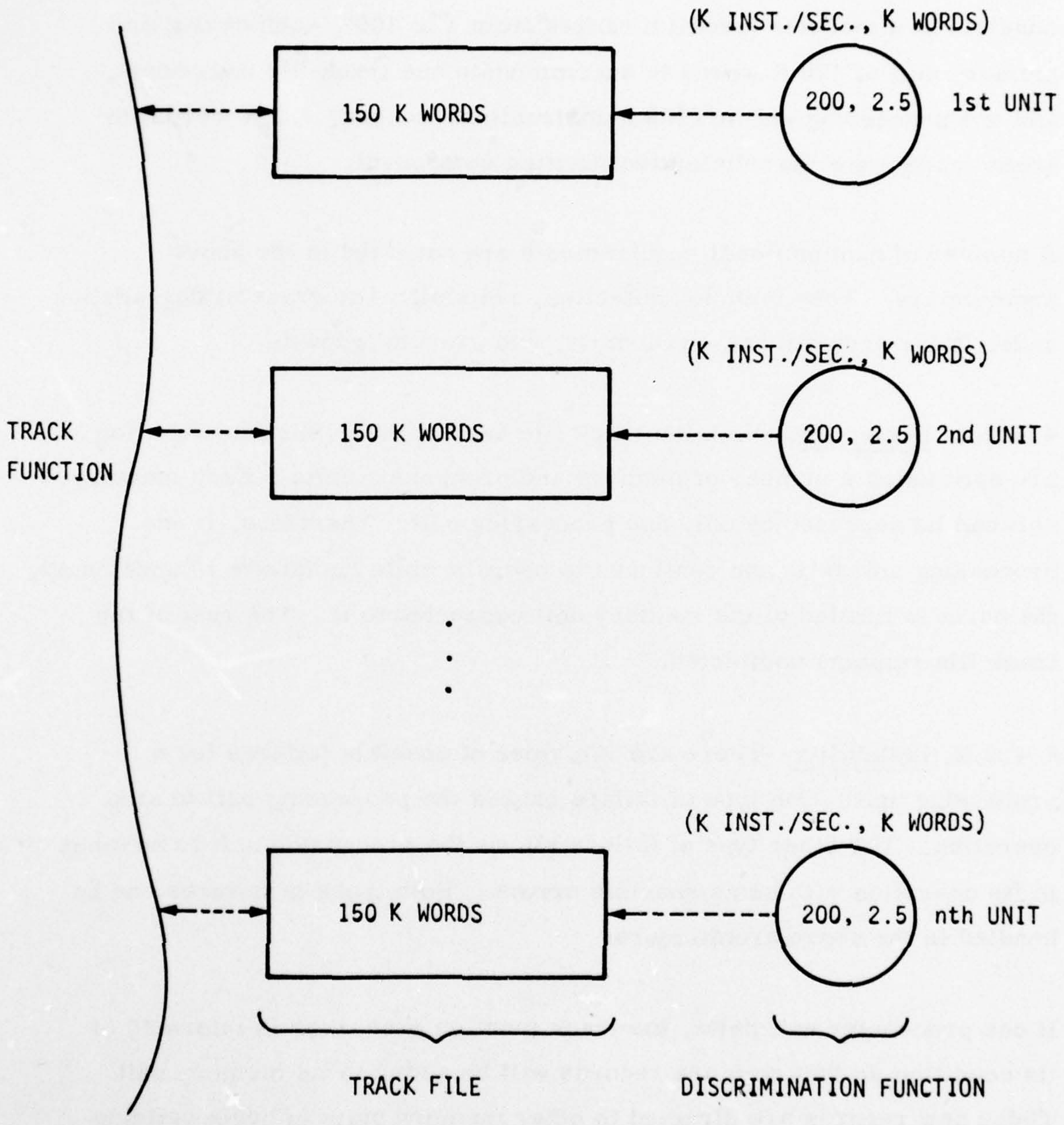


Figure 49. An Architecture Graph for the Discrimination Function and the Track File

to be informed of the condition of the processing units, each unit is expected to send (periodically) some messages to the track function, via the memory unit between them. If the track function has not received a message from a processing unit for some time, it recognizes that this processing unit has died. It then eliminates this module from the system by not sending any new records to it until it is repaired.

To handle the second type of failures, each processing unit should have some self-checking capabilities.[32, 33, 34] (The processing and memory requirements of these capabilities are not considered in the architecture graph of Figure 49.) If a processing unit detects an error in its own behavior, it halts, thereby causing the track function to eliminate it from the system until it is repaired.

To handle memory errors, each processing unit checks its memory unit regularly. (The processing and memory requirements for these checks are not considered in the architecture graph of Figure 49.) On detecting a memory error, the processing unit halts, causing the track function to eliminate it until it is repaired.

4.4.2.3 Uniformity--The architecture is uniform in the sense that it uses only two kinds of modules: memory units of 150 K words, and processing units (200 K instructions, 2.5 K words). The building blocks are interconnected in a uniform way with a small number of interconnections.

4.4.2.4 Graceful Growth--The architecture grows linearly with n, where n is the expected number of enemy threat clouds handled by the discrimination function. The "growth step" is 1 threat cloud. This means that the architecture can grow to handle n + 1 threat clouds, instead of n, by adding one module (containing one memory unit and one processing unit).

To increase both the protection and the growth flexibility, each module in the architecture graph of Figure 49 can be decomposed into two modules as shown in Figure 50. This new architecture satisfies all the nonfunctional requirements satisfied by the previous architecture. Moreover, it provides more protection since its track file is divided into more components, so its growth step is ½ threat cloud which increases the growth flexibility.

To establish the credibility of the above architecture, we consider the problem of designing a compatible computer architecture for the track function. This is done in the next section.

4.4.3 An Architecture for the Track Function

Figure 51 shows the architecture graph of Figure 50 with the track function implemented in a set of processing units. Each of these units is assigned to one discrimination module. One processing unit, the distributor, takes records from the new frames file and sends them over the bus to track function processing units.

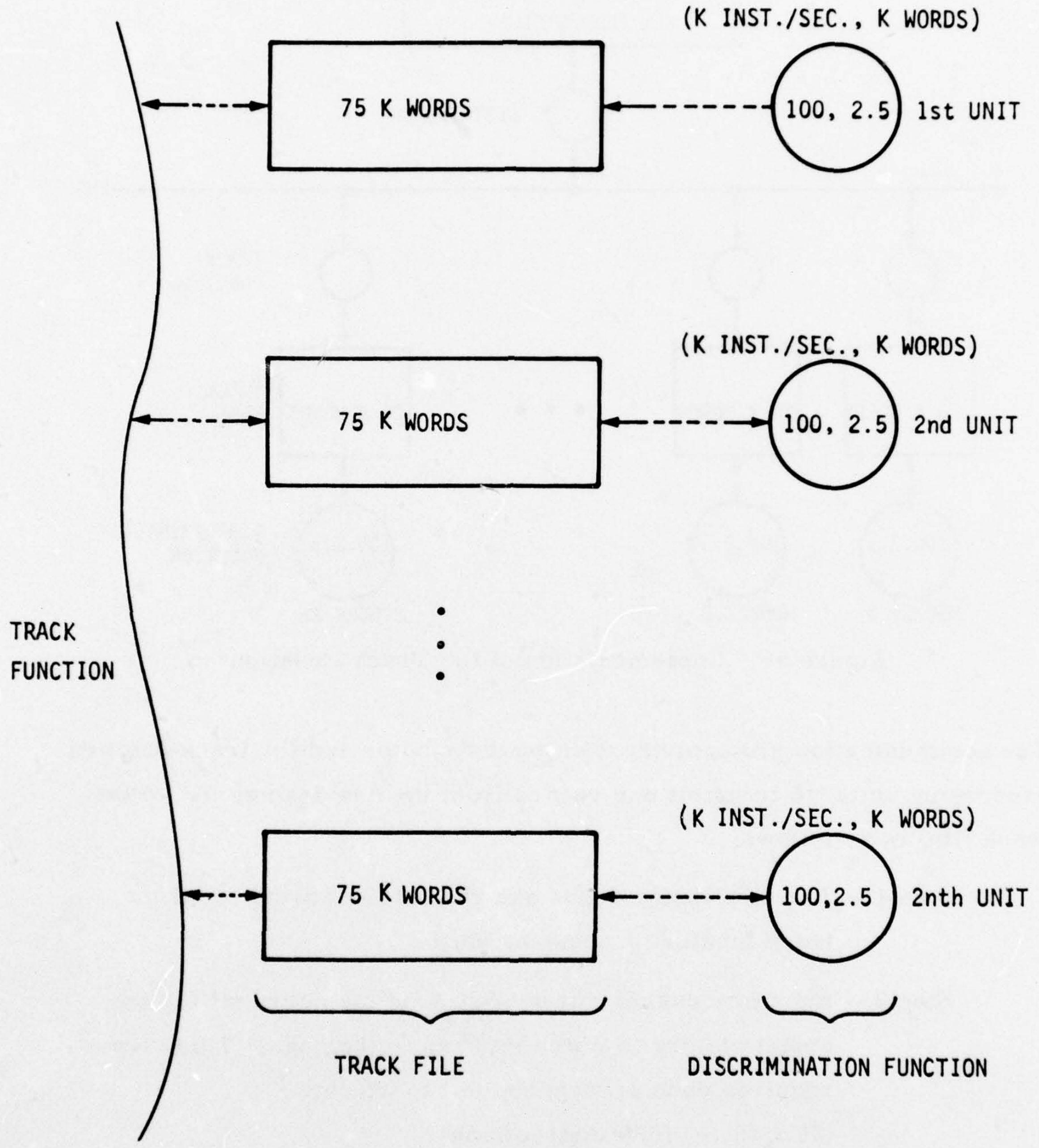


Figure 50. Another Architecture Graph for the Discrimination Function and the Track File

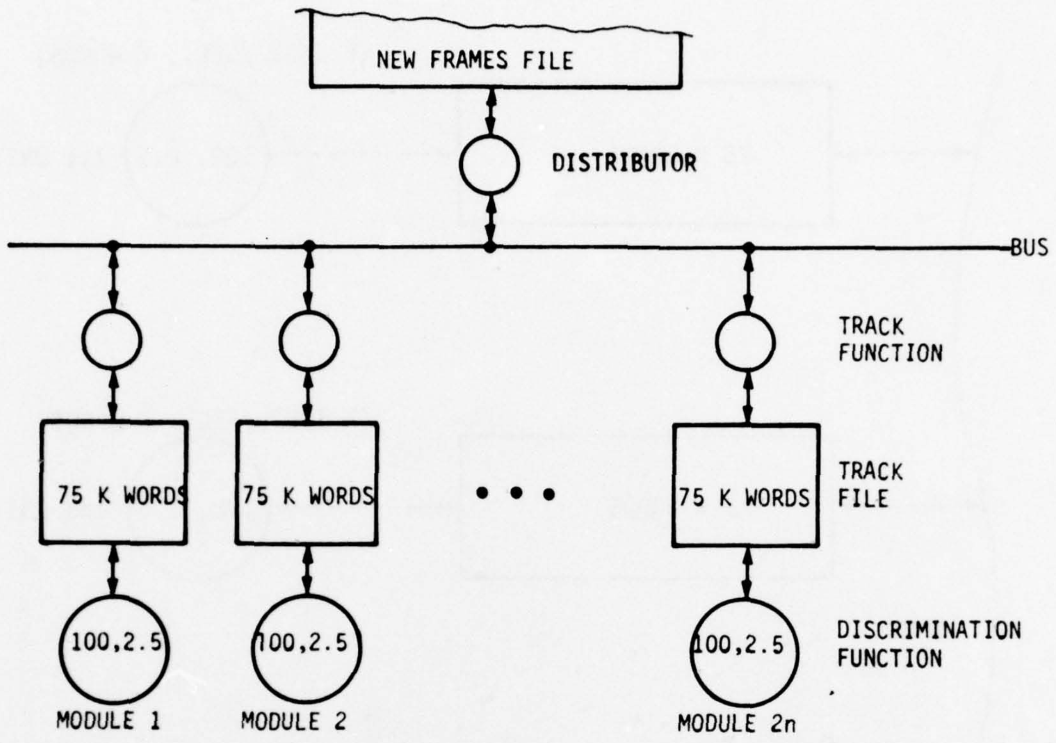


Figure 51. Implementation of the Track Function

The communication protocol between the distributor and the track function processing units (to transmit one record from the new frames file to the track file) is as follows:

Step 1-- The distributor sends one record (20 words) to all the track function processing units.

Step 2-- Each processing unit searches its memory unit for the object history to which this record belongs. This step requires each processing unit to execute ($36 \times 25 < 1000$ instructions*).

*Recall from Section 4.4.1 that the number of instructions executed by a track function is 36 h, where h is the number of object histories in the track file unit.

Step 3-- Each processing unit informs the distributor whether or not such an object history has been found in its memory unit. All these answers from the processing units can be "multiplexed" into one word transmission since the distributor needs only to know whether or not such an object history has been found.

Step 4-- If no such object history is found, which implies that the record belongs to a new object history, the distributor asks one processing unit to store the record in its memory unit as the beginning of a new history.

The activities carried out by each track function processing unit according to this protocol are outlined in Figure 52. Since records are added at the rate of 200 n records/second with n ranging from 1 to 100, the worst loading condition occurs when n = 100. In this case, 22 words need to be transmitted and 1000 instructions need to be executed within 10^{-4} seconds. If we divide this time period between these two activities as follows,

0.1×10^{-4} seconds to transmit 22 words,

0.9×10^{-4} seconds to execute 1000 instructions

we can compute

$$\text{Bus transmission rate} = \frac{22}{0.1 \times 10^{-4}} = 2.2 \text{ M words/second}$$

Processing rate for one track function processing unit

$$= \frac{1000}{0.9 \times 10^{-4}} = 11.1 \text{ M Instructions/second}$$

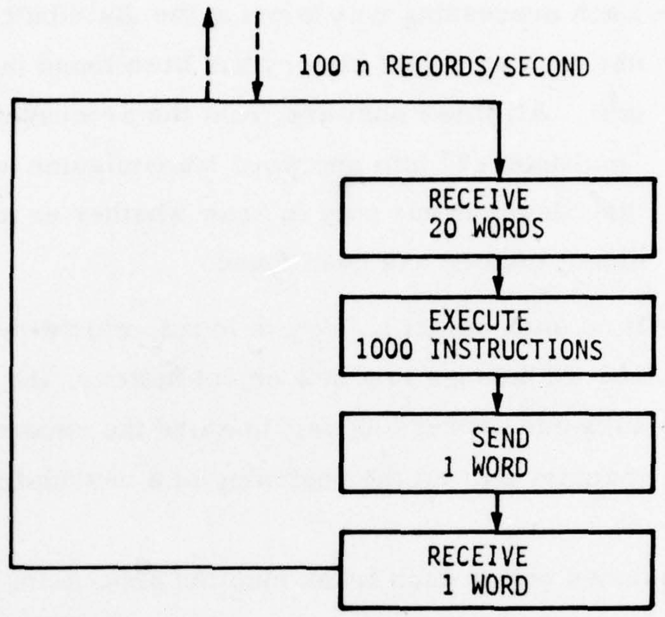


Figure 52. Outline of the Track Function Processing

These numbers are too high, so another architecture is needed to implement the track function in the worst loading condition. The new architecture is shown in Figure 53. It is similar to the one in Figure 52 except that it has four buses and the track function is implemented in four processing units per module. In this case, the distributor sends the records one by one cyclically over the four buses as follows. The first record is sent over the first bus; the second record is sent over the second bus; the third record is sent over the third bus; the fourth record is sent over the fourth bus, and the process repeats.

Over one bus, two successive records arrive within 4×10^{-4} seconds. In this time period, 22 words need to be transmitted and 1000 instructions

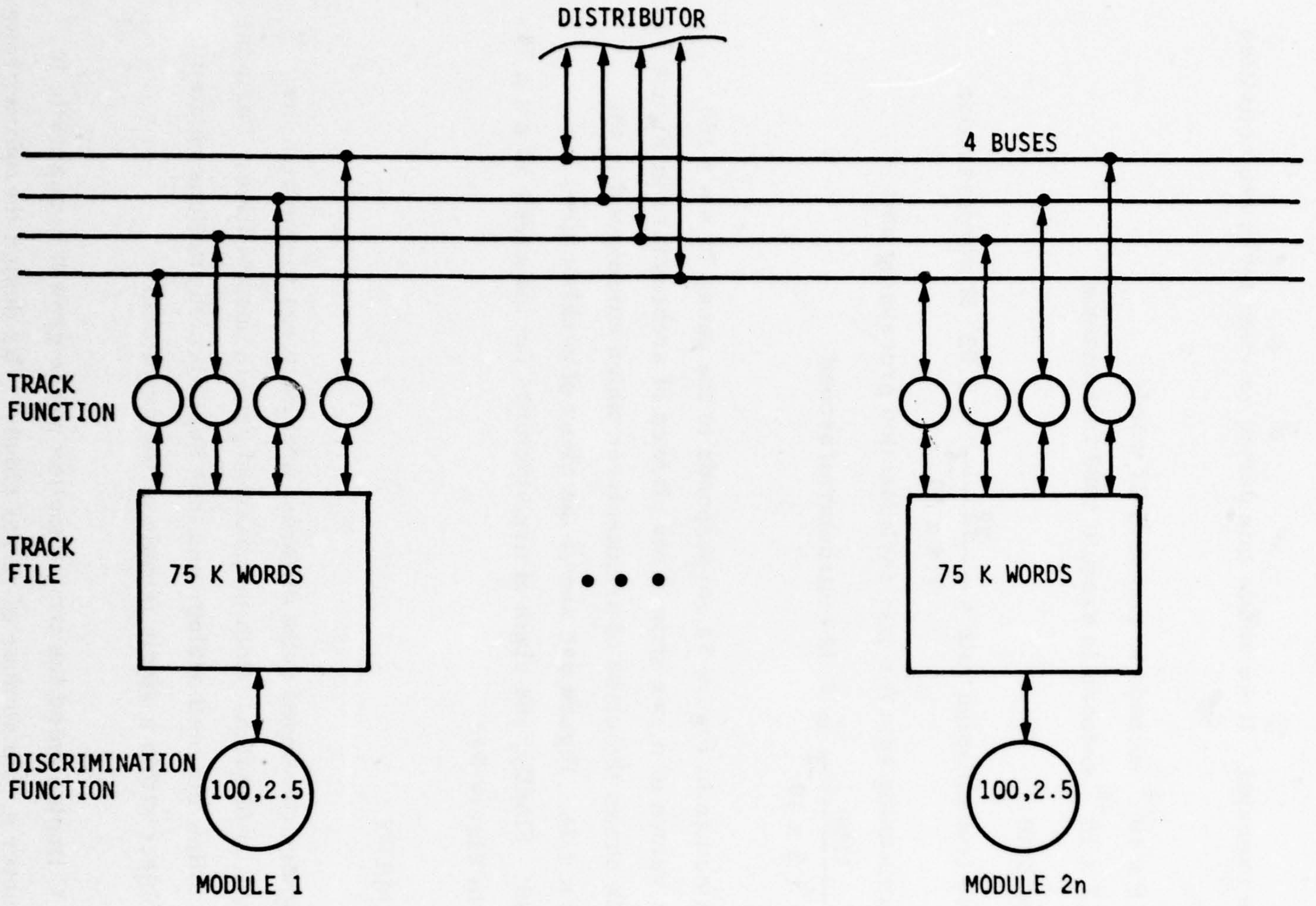


Figure 53. Outline of the Final Architecture

need to be executed. If we divide this period between these two activities as follows,

0.8×10^{-4} seconds to transmit 22 words

3.2×10^{-4} seconds to execute 1000 instructions

we can compute

$$\text{Bus transmission rate} = \frac{22}{0.4 \times 10^{-4}} = 0.55 \text{ M words/second}$$

Processing rate for one track function processing unit

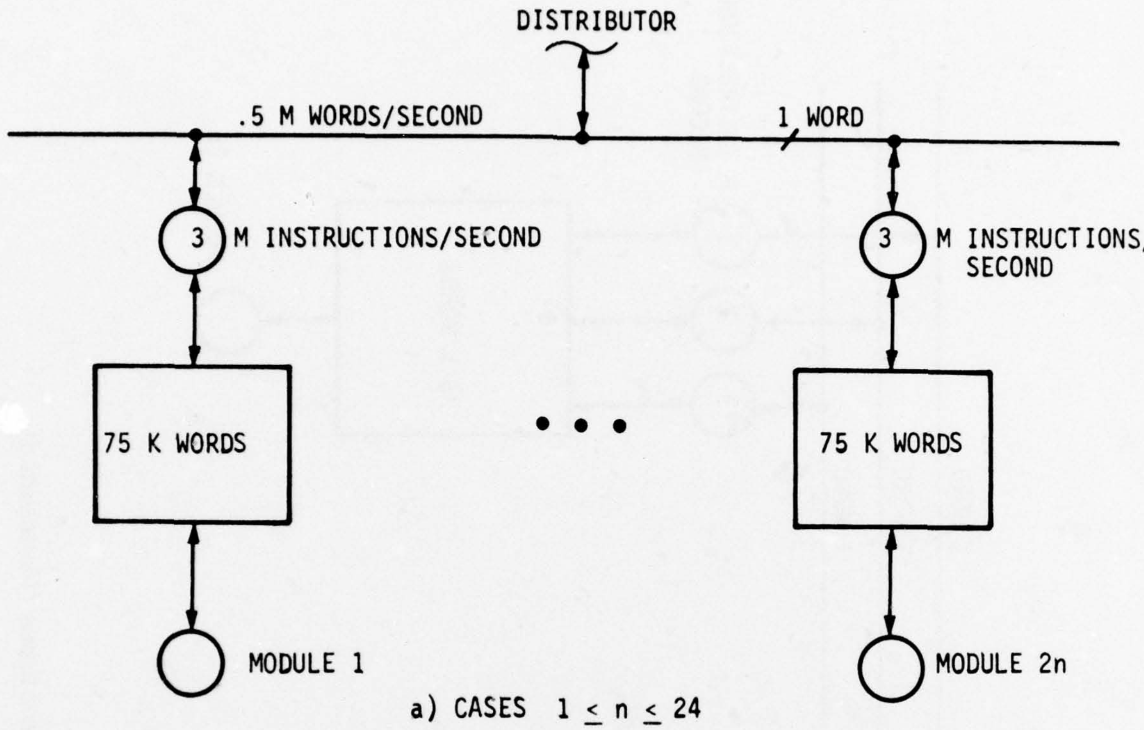
$$= \frac{1000}{3.5 \times 10^{-4}} \approx 3 \text{ M instructions/second}$$

The architecture in Figure 53 corresponds to the cases $75 = n = 100$. For other values of n, we show three classes of architecture in Figure 54. Figure 54a shows the class of architectures which correspond to the cases $1 \le n \le 24$. Figure 54b shows the class of architectures for $25 \le n \le 49$. Finally, the class of architectures for the cases $50 \le n \le 74$ is shown in Figure 54c.

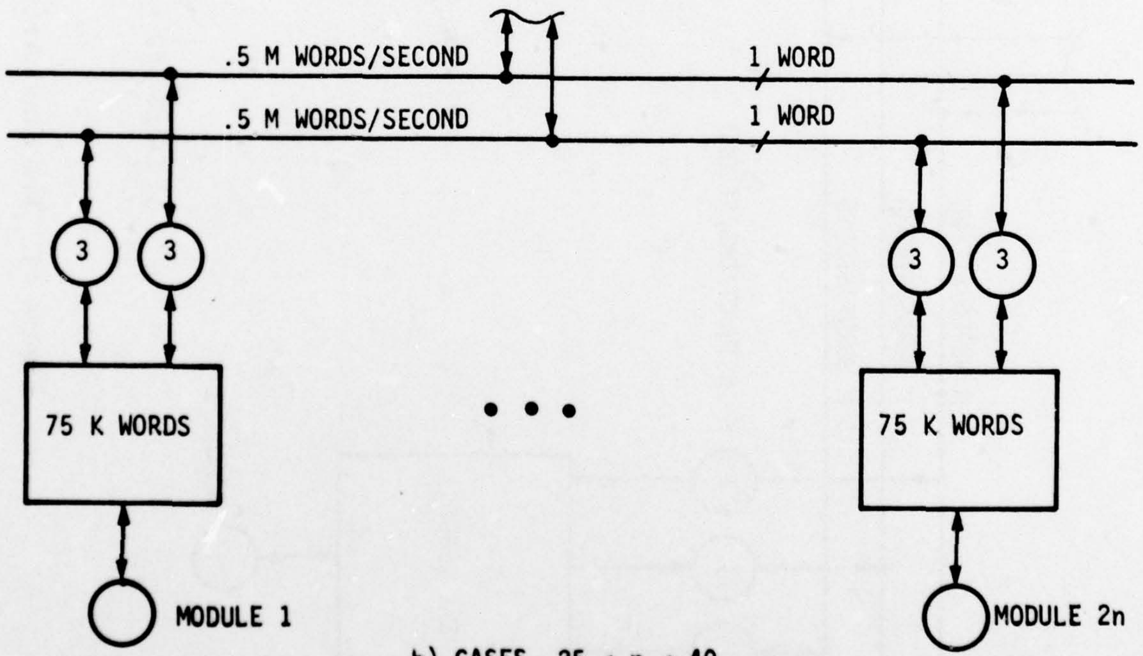
4.5 SUMMARY

Two designers developed quite different architectures to perform the discrimination function. Both emphasized growth possibilities. Designer C did not follow his methodology and thus kept all computations related to a single object within a single module of the architecture.

Designer C implemented the track function whose growth is quadratic in the parameter n, the number of threat clouds. His design therefore grows

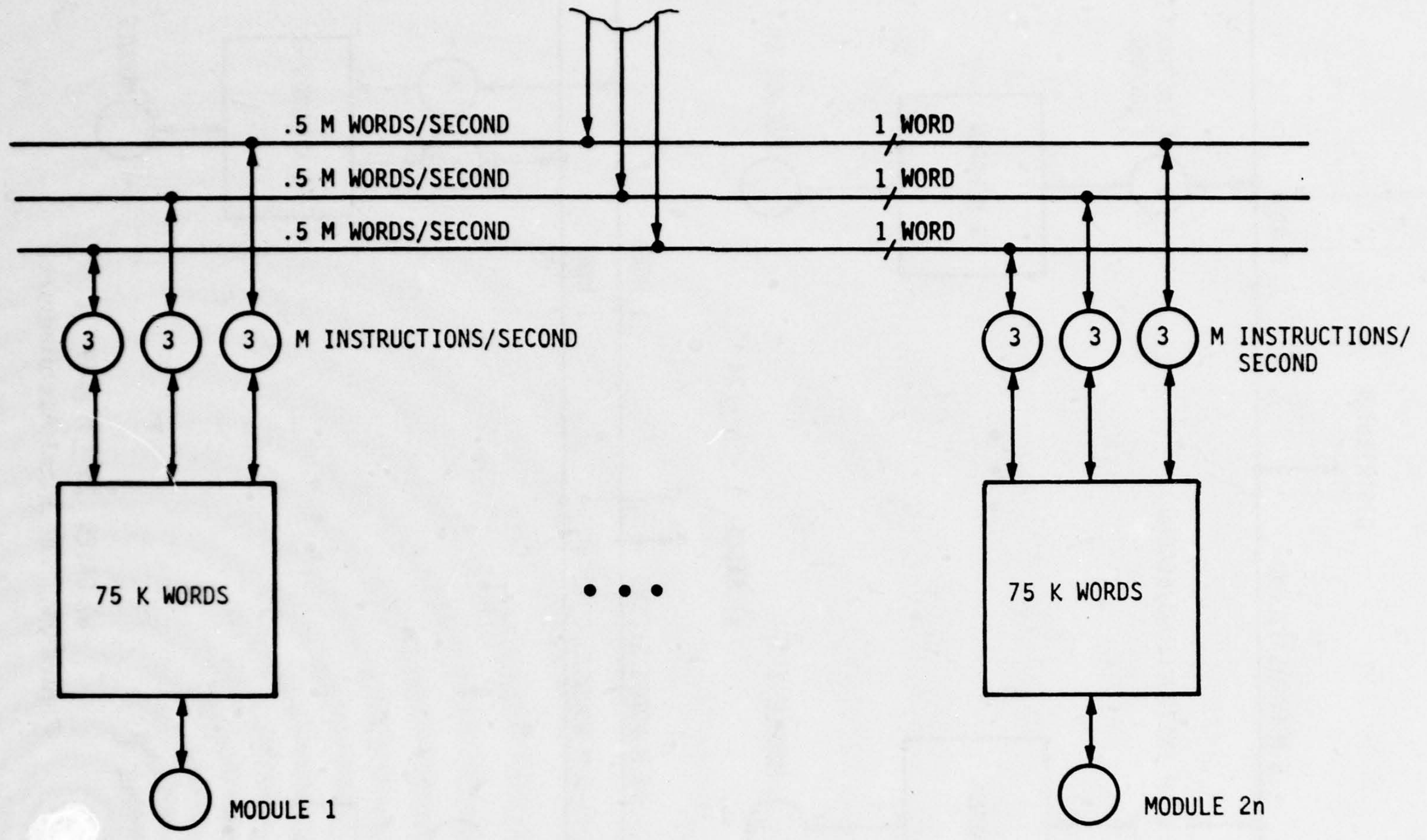


a) CASES $1 \leq n \leq 24$



b) CASES $25 \leq n \leq 49$

Figure 54. The Final Architecture



c) CASES 50 $\leq$ n $\leq$ 74

Figure 54. The Final Architectures (Concluded)

in two ways as n increases; the number of memory and discrimination modules increases and the number of track modules associated with each memory module also increases.

Reliability and degradation issues were considered by designer E but not by designer C. Designer E's concern was primarily with communications required to pass the load from failed modules to well modules.

A number of areas for further study were uncovered during this design exercise; they include:

- Problem specifiers must state policies concerning the system's response to saturation caused by many inputs.
- Control interfaces among functions have not been specified.
- Functional interface specifications--how parameters and results are passed among functions--strongly affect how the functions are implemented.

It is clear that the function specifiers must be worried about these issues or else they must leave the interfaces to be negotiated among designers of interacting modules.

These issues are discussed further in Sections 6 and 7.

SECTION 5

NONFUNCTIONAL REQUIREMENTS OF DISTRIBUTED COMPUTER SYSTEMS

Many different distributed systems can be constructed to meet any given set of functional requirements (e.g., processing rates, memory). The designer selects one of these systems based on how well each system meets another set of nonfunctional requirements. Usually, the set of nonfunctional requirements includes cost, reliability, modularity, and "easiness" of growth. Unfortunately, there exist neither standard definitions for these terms, nor standard techniques to estimate and compute their values.

In this section, we consider two of these nonfunctional requirements and discuss how to incorporate them in "good" architectures under any design methodology. The two chosen requirements are graceful degradation (Section 5.1) and graceful growth (Section 5.2).

5.1 GRACEFUL DEGRADATION

Graceful degradation of a distributed system is the property that, if some components in the system fail, the other components can detect these failures and compensate for them and the system continues to operate properly, probably with lesser accuracy and/or at a lesser rate.

Figure 55 shows a cascade architecture with three processing units. Unit i ($1 \leq i \leq 3$) performs a function f_i on its input data stream before producing its output data stream. If any of the three units fails, then the function of this unit disappears from the system, and the path from the external input to the external output becomes disconnected.

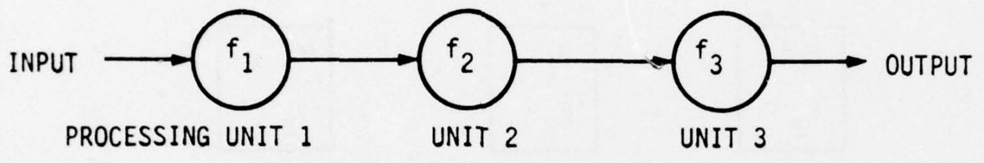


Figure 55. Cascade Architecture with Three Processing Units

To handle function loss, each function (or, at least, a simplified version of it) should be prestored in each processing unit in the system. This can be achieved by associating a backup store with each processing unit; the backup store contains versions of all functions in the system, as shown in Figure 56.

Now we need a mechanism to compensate for the loss of the path between the external input and the external output. This is achieved by adding a bypass connection around each processing unit to connect its input directly to its output, as shown in Figure 56. The bypass connection is activated only if the associated processing unit becomes nonoperational.

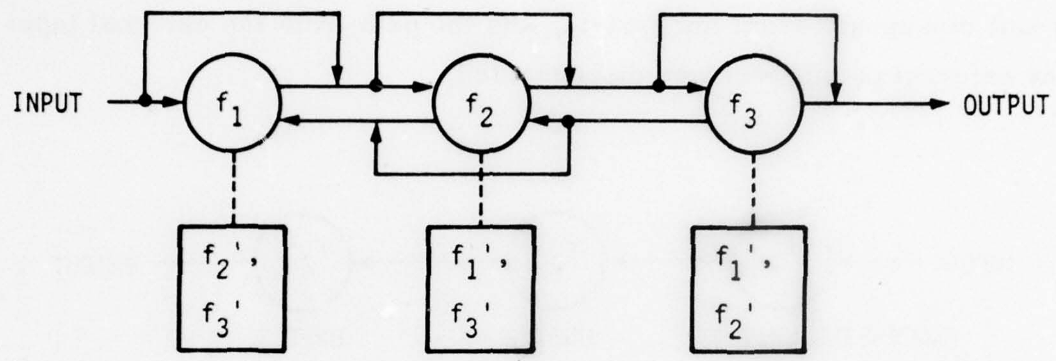


Figure 56. Cascade Architecture of Figure 55 Modified to Improve Graceful Degradation Characteristics

The processing units should exchange control messages so that each unit knows the current status of all other units in the system. This exchange may require extra connections between processing units to carry these messages. For example, in Figure 56 "backward" connections are added from unit 3 to unit 2 and from unit 2 to unit 1; a backward bypass connection is added around unit 2.

The above scheme can be generalized to a wide class of distributed systems with point-to-point connections. Distributed systems with multi-point connections like buses and loops need further attention. As an example, Figure 57 shows a bus architecture with three processing units. The external input data stream goes to processing unit 1 which executes function f_1 on it; it then uses the bus to send the results to unit 2. Unit 2 executes f_2 and sends the results to unit 3 (using the bus). Finally, unit 3 executes function f_3 on the data and sends the results as an output of the computation.

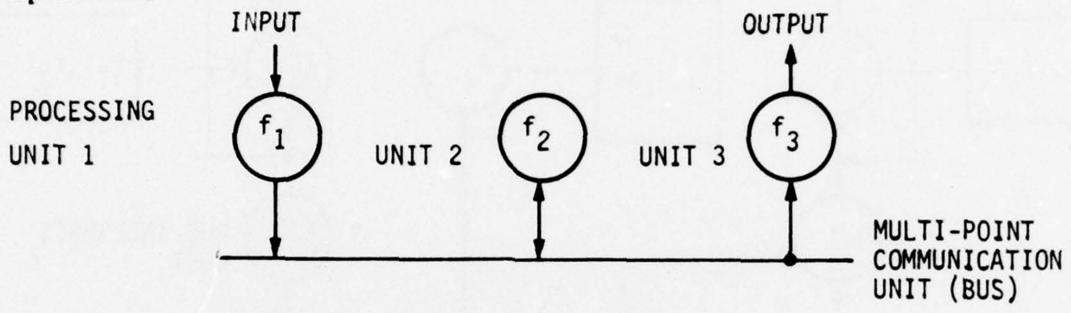


Figure 57. Bus Architecture with Three Processing Units

In this case, simple bypass connections are not sufficient to handle the loss of the path from the input to the output. Bus interface units, needed to control the communication over the bus, are part of the critical path from input to output. For this reason, bus interface units should be separated from the processing units (at least in the cases of unit 1 and unit 3 as shown in Figure 58).

5.2 GRACEFUL GROWTH

Graceful growth requirements should be considered in an early stage of the design process because systems that are built without considering possible growth are not easy to modify afterwards to meet growing demands. We limit the discussion in this section to planned (predefined) growth.

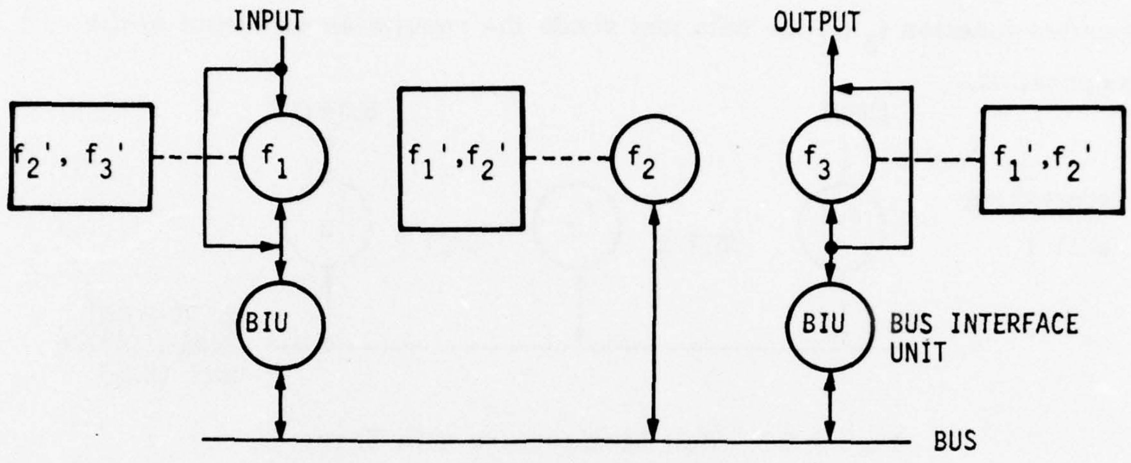
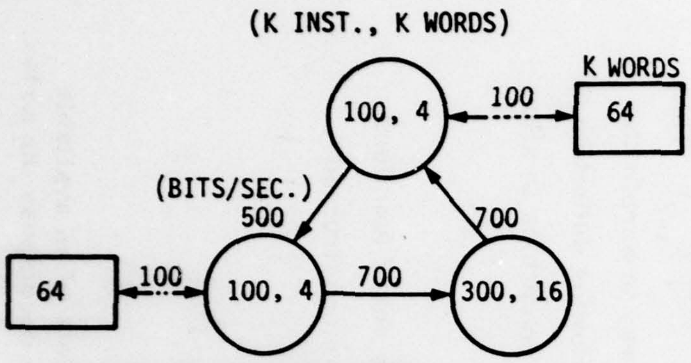


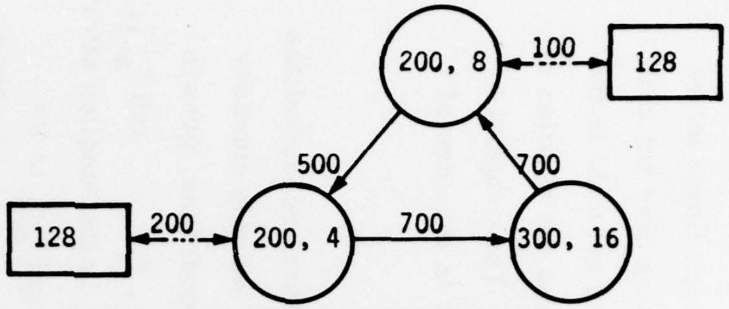
Figure 58. Bus Architecture of Figure 57 Modified to Improve Graceful Degradation Characteristics

Growth of distributed systems can be achieved by any of three ways: growth in size, growth in function, and growth in both size and function. Consider the example in Figure 59a, which shows an architecture graph* describing a distributed system. It contains three processing units, two memory units, and three communication links. For this architecture graph to grow in size, the graph structure remains as it is, and some of its labels grow in value, as shown in Figure 59b. For this architecture graph to grow in function, it is embedded in a larger labeled graph but its labels remain as they are, as shown in Figure 59c. Figure 59d shows the architecture graph after some growth in both size and function.

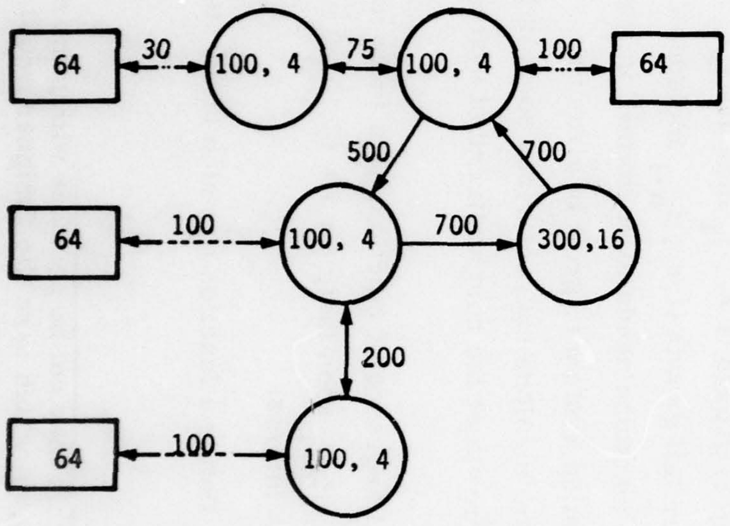
*Architecture graphs are defined in Section 2.1.2.



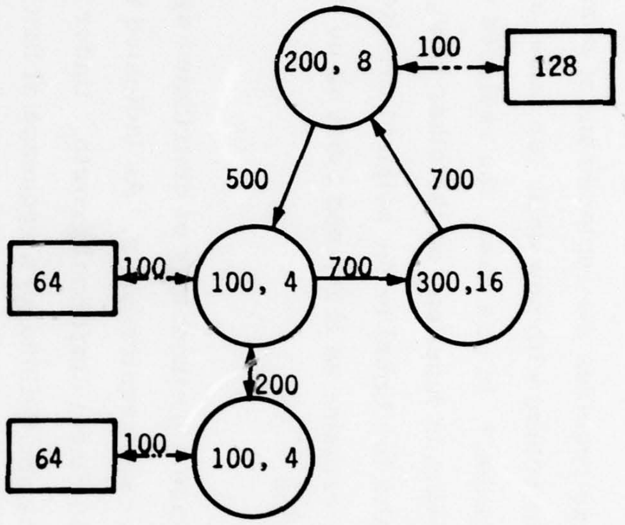
(a) AN ARCHITECTURE GRAPH



(b) THE ARCHITECTURE GRAPH AFTER SOME GROWTH IN "VALUE"



(c) THE ARCHITECTURE GRAPH AFTER SOME GROWTH IN "STRUCTURE"



(d) THE ARCHITECTURE GRAPH AFTER SOME GROWTH IN BOTH VALUE AND STRUCTURE

Figure 59. The Three Types of Growth

Growth in size requires the replacement of some units (or links) in the original architecture with new units (or links) of the same type but with larger capacities.* In this case, the replaced units are not used in the system. Growth in function, on the other hand, requires the addition of some new units (or links) to the original architecture. The original architecture remains as it is, and there is no waste as in the case of growth in value.

We now discuss a methodology of distributed systems that accommodates functional growth requirements. As indicated before, we only consider planned (predefined) functional growth. Under these conditions, growth requirements are defined as a sequence of functions f_o, f_1, ..., and f_{n-1}, where f_o is the original function to be implemented, f_1 is the function after the first step of growth (i.e., f_1 includes f_o), ..., and f_{n-1} is the function after full growth (i.e., f_{n-1} includes f_1, f_2, ..., f_{n-2}). This sequence of functions needs to be defined in a special way to permit the host distributed system to grow in structure. The following definition of a sequence of functions characterizes some sufficient conditions to allow structural growth of the host distributed system.

A structural sequence of functions is a finite nonempty set of functions f_o, f_1, ..., f_{n-1} such that, for $i = 1$, ..., $n-1$, f_i is constructed from f_{i-1} as follows:

1. Define a function c_i which interacts with f_{i-1}.

*Note that this may not be possible within the constraints of an available technology, in which case the designer would have to decompose the nodes.

2. Modify f_{i-1} to interact with c_i. Let f'_{i-1} be the function f_{i-1} after modification.

3. In this case, function f_i consists of the two interacting components c_i and f'_{i-1}.

As an example, Figure 60 shows a structural sequence $\{f_o, f_1, f_2\}$ of functions. Function f_o, shown in Figure 60a, is represented as the cyclic function c_o (see Section 2.1 for a description of cyclic functions). Function f_1, shown in Figure 60b, is represented as the two interacting cyclic functions c_o' and c_1. Finally, function f_2, shown in Figure 60c is represented as the three interacting cyclic functions c_o'', c_1', and c_2.

The notation $\{f_o, f_1, \ldots., f_{n-1}\}$ is adopted for any structural sequence of functions. Function f_o has one component c_o; f_1 has two components $c_o^{(1)}$, and c_1^o; ...; f_{n-1} has n components called $c_o^{(n-1)}$, $c_1^{(n-1)}$, ..., $c_{n-2}^{(n-1)}$, and c_{n-1}. These components are not necessarily "basic" components since any of them can be decomposed into "smaller" components.

The required design methodology operates on any given structure of functions $\{f_o, f_1, \ldots., f_{n-1}\}$ and produces a structure of "good" architectures $\{a_o, a_1, \ldots., a_{n-1}\}$ such that

1. For $i = 0, 1, \ldots, n-1$, architecture a_i corresponds to function f_i.

2. For $i = 0, 1, \ldots, n-2$, architecture a_{i-1} is constructed from architecture a_i after some structural growth.

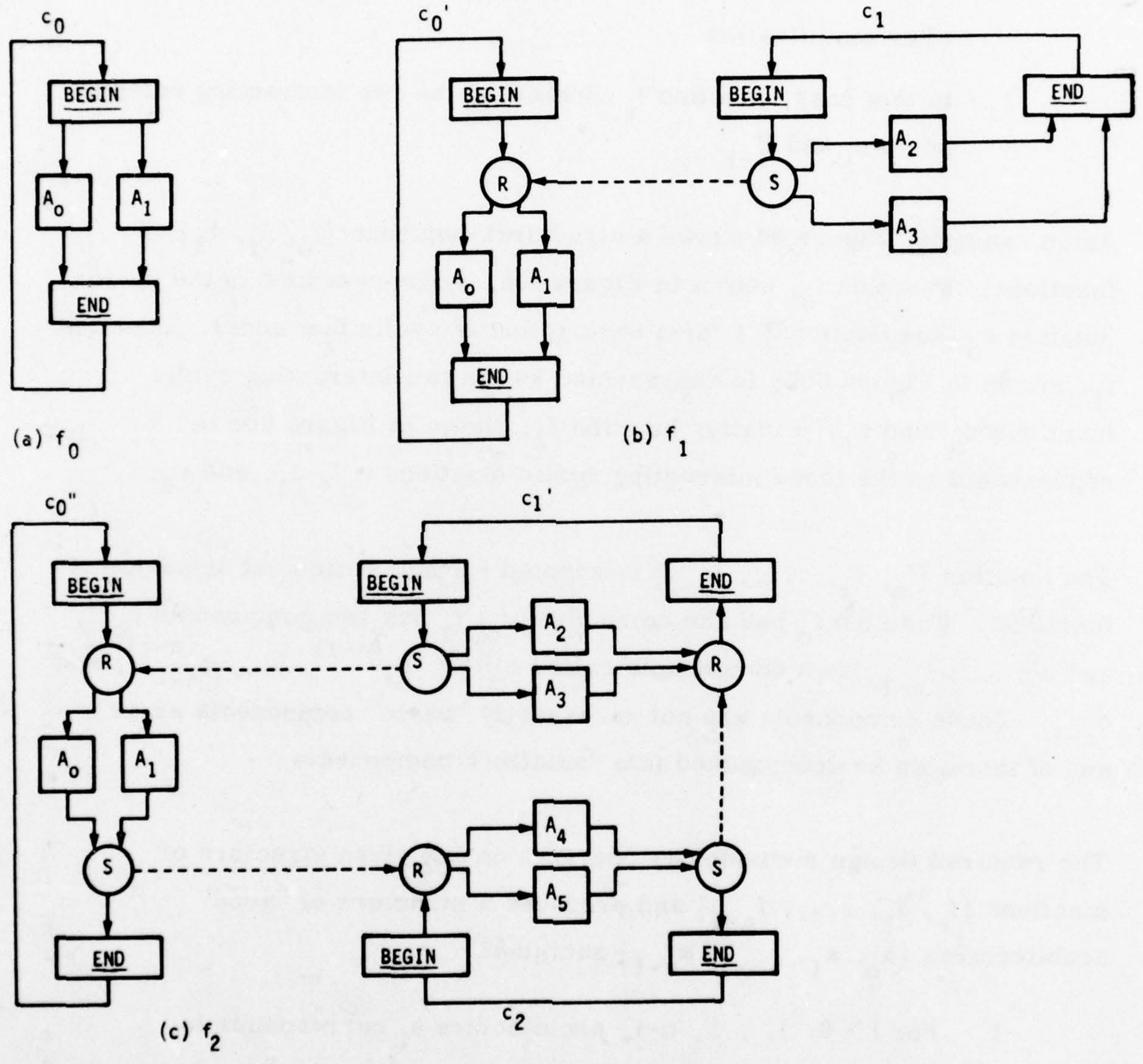


Figure 60. A Family of Three Functions

One design methodology can be summarized as follows:

```
for  $i = 0, 1, \ldots, n-1$ do:
begin   input $c_i^{(n-1)}$
        get a "good" distributed architecture $a_i$ for $c_i^{(n-1)}$
            comment use any known design methodology which
            does not consider growth requirements. Two
            candidate methodologies are defined in Sections 2.1
            and 2.2 of this report
            end of comment
        output $a_i^{(n-1)}$
end
```

Some comments on this methodology are in order. First the output architectures do not correspond directly to the functions in the given structure. However, the correspondence can be defined as follows:

a_o^o corresponds to f_o,

<u>connect</u> (a_o^1, a_1^1) corresponds to f_1, ...

<u>connect</u> $(a_o^{(n-1)}, a_1^{(n-1)}, \ldots\ldots, a_{n-1}^{(n-1)})$ corresponds to f_{n-1}.

The operator <u>connect</u> takes a number of architectures as an input and produces an architecture resulting from connecting them together. Combinations such as connect (a_o^2, a_1^2) produce implementations of functions (f_1, in this example) including all the synchronization and communication operations needed to connect f_1 into a larger function.

A growing architecture could be implemented in several ways from these implementations a_i^j. The designer could decide to implement f_1 using any of the a_1^j architectures. When the system grows, he simply connects more a_i^k structures to implement the additional functions. When he tries to extend the system beyond the functions for which the synchronization and communication "hooks" have been included, he encounters some difficulty because he must redesign the previous portions to accommodate these new needs. The question of how to optimize life cycle costs in a system growing in this manner has not been studied under this contract.

Finally, note that the full architecture implemented by the algorithm suggested above is not necessarily the optimum realization of the overall function since the only optimizations were within the incremental additions considered separately.

SECTION 6

SUMMARY

The efforts described in this volume included selection and study of two BMD functions, development of architectures to implement these functions, preliminary specification of methodologies useful for developing architectures from functional specifications, and specification of some design techniques useful to improve the characteristics of architectures. In Section 6.1 we summarize our efforts very briefly. The major content of this section is in Section 6.2 where we emphasize the observations concerning these efforts; our suggestions for further research and development activities and experiments are contained in Section 7.

6.1 EFFORT SUMMARY

6.1.1 Architectural Development for BMD Functions

We studied the radar scheduling and optical discrimination functions and developed several architectures to implement each function, as described in Sections 3 and 4. Four designers with different experience developed architectures for the radar scheduling function. In addition to discovering the specific architectures, we learned about development methodologies and function specification difficulties. The later development of functional specifications for the optical discrimination function was influenced by these difficulties. As a result, we did find a better specification of the discrimination function for which two designers developed architectures. Needs for better techniques to describe sequencing and implementation of specific functionality within specific hardware modules were uncovered.

6.1.2 Architectural Design Methodologies

Two of the radar scheduler architecture designers developed and documented architectural design methodologies. Though superficially similar, the two methodologies differ in their emphases on how sub-function implementations are distributed among hardware modules. It seems that the two techniques can be merged to form a more comprehensive methodology useful for several levels of design detail.

6.1.3 Design Evaluation Techniques

One designer collected some specific design techniques for improving the system reliability. More work in similar directions would lead to evaluations and improvements of architectures with respect to other important payoff attributes.

6.2 OBSERVATIONS

We will summarize the lessons we learned and other observations concerning these design efforts. The discussion is divided, not according to the phases of the effort in this project, but according to aspects of specification and design tasks. We start with function specification techniques in Section

6.2.1, continue in Section 6.2.2 with design constraint specifications, turn to the design steps themselves in Section 6.2.3, and discuss architectural description techniques in Section 6.2.4. A parallel structure in Section 7 includes suggestions for further research and development efforts based on these design experiences.

6.2.1 Function Specifications

The following observations are derived from our experience with specifying the radar scheduling and optical discrimination functions.

6.2.1.1 Function Specificity--If different designers are to develop architectures to be compared, they must be implementing a specific, well-defined function. However, see Section 6.2.1.3.

6.2.1.2 Basis for Functional Specification--Many BMD functions involve sets of objects. Functions dealing with such sets may be specified by giving the actions to be taken on a single object or by giving the actions upon the complete sets. A "kernel" function describing the actions on a single object must be supplemented by a specification of object interactions within the function processing. Object-based specifications may be appropriate when pipeline or array processing may be useful (see Section 6.2.1.10).

6.2.1.3 Implementation Independence--We want a function description which is independent of possible implementations. As noted in Section 3.1, it is difficult to obtain such descriptions from those knowledgeable at the systems level of the problem. Of course, it is quite difficult to allow design freedom yet to use a specific function (see Section 6.2.1.1).

6.2.1.4 Permissible Parallelism--We noted that the function specifier needs a technique to specify parallelism that may be used in the implementation. Specifically, options for array or pipelined processing must be noted. (It is much more efficient for the specifier to state parallelism when he knows it is possible.)

6.2.1.5 Specification Communication--We experienced difficulties when designers attempted to communicate functional specifications among themselves. One consequence of this communication problem was that three different design problems were solved for the radar scheduling problem.

6.2.1.6 Decision Probability Specifications--One reason for different radar scheduler designs was the lack of a clear specification of the probability of constraint conflict. This parameter turned out to play a significant role in the architectures, but that fact was not apparent until several designs were contrasted. A clear specification of such important problem parameters is essential. Feedback between the function specifiers and the architecture designers will be necessary to clarify such hidden problems.

6.2.1.7 Processing Requirements--The function specifier must give some information concerning the processing requirements associated with each significant subfunction used to implement the function so that the designer can size the processing element.

6.2.1.8 Memory Requirements--The function specifier must give information concerning the amounts of memory required for temporary storage and for permanent data bases associated with each subfunction so that the designer can configure memory elements appropriately.

6.2.1.9 Performance Requirements--The function specifier must give the required timing of the function so that the designer may make appropriate trades in his design process. It is difficult to do this in an implementation-independent manner. The following three points are related to this one.

6.2.1.10 Input Arrival Specifications--The real-time system designer requires information concerning the rates and patterns of the arrival of input information (or requests). Furthermore, he must know how the function is to respond to the unexpected arrival of an input. This information must be included in the function specification.

6.2.1.11 Saturation Specifications--The function specifier must inform the designer how the function should respond when the input arrival rate exceeds the processing abilities of the system. Designers must make arbitrary decisions regarding queueing the additional requests, dropping some requests, or attempting to service all with a longer delay, if they are not informed concerning this important issue. Since the success or failure of a real-time system may depend on its saturation behavior, the specifier must provide guidance.

6.2.1.12 Degradation Specifications--Systems may lose capabilities for numerous reasons. When the capabilities are degraded, the performance diminishes. If the system should simply discard requests that it cannot handle, then the architecture is not affected by the need to provide degraded performance. If, however, the system should substitute alternate functions under degraded conditions, the designer might consider how to implement the degradation alternatives efficiently. A specification limiting the designer's scope in this matter should be provided.

6.2.1.13 Parameter Ranges--We find that designers have trouble using parameters that are obviously average estimates. Function specifiers should provide either ranges within which the system must deliver specified performance or probability distributions of the parameter values.

6.2.1.14 Function Interface Specifications--Our experience with the discrimination algorithm shows that the interface between that function and the track function is important in determining efficient implementations of each function. Designers of communicating modules must coordinate the interface description for efficient system design.

6.2.2 Design Constraints

We observed that there was no clear designation concerning which system attributes should be emphasized by the designers. Furthermore, they not only need requirements for these attributes, but they also need reasonable requirements that they may impose on the modules used in their designs.

6.2.2.1 Which Attributes--The design problem specifier must state which system attributes shall be emphasized by this design iteration. If the attributes can be quantified, this specification could be in the form of a benefit function expression and a minimum value of that function that is acceptable.

6.2.2.2 Component Attributes--Designers cannot guarantee system attributes without assumptions regarding the attributes of the system's components. The specifier of the design task must state which attribute combinations may be assumed for the system's components; these statements will clarify the designer's task.

6.2.2.3 Taxonomy--Designers would be assisted by a taxonomy of techniques that could be used to achieve each desired system attribute. Individual designers would not have to "reinvent" design techniques to achieve certain attributes if they could refer to a taxonomy and make an appropriate selection.

6.2.2.4 Growth--The growth attribute received particular attention during these studies. We found a lack of useful ways to specify growth behavior; the designers wanted to know not only how much maximum growth is required, but also how much price can be paid to achieve that growth.

6.2.3 Design Steps

These observations are based on the new design methodologies, the experience of the designers, and our reflections on all these developments.

6.2.3.1 Documentation--Designers must document the bases for their design decisions as well as the decisions themselves. A standard format should be developed to guarantee appropriate documentation.

6.2.3.2 Payoff Emphasis--Designers must clearly state the trades they make among payoffs as they document design decisions. Persons evaluating designs should consult this information; it may suggest modifications to accommodate changed specifications or revisions of payoff emphasis.

6.2.3.3 Tradeoff Quantification--Just as designers must know how to evaluate their designs, so also must they know how to quantify design trades. When several alternatives have similar values, perhaps all should be considered until more information is developed in further design steps.

6.2.3.4 Methodology Comparison--The two methodologies describe the system at different levels of detail: one emphasizes distributing subfunctions across all modules in the design, while the other distributes the function among its subfunctions with each subfunction performed by a separate module. These techniques can be interleaved to perform designs through all system levels.

6.2.3.5 Architectures for Reliability--A number of architectures were developed specifically to improve system reliability; several different interconnection techniques were used. We need more study concerning the values of these techniques.

6.2.3.6 Processing Emphasis--All designers emphasized the processing requirements while developing methodologies and designs. Perhaps this was a consequence of the functions selected for study.

6.2.3.7 Validation and Verification--No designer made any attempt to verify or validate any design step. Design efforts for large systems require such activities.

6.2.4 Description Techniques

The following observations concern the operation of systems created by the designers. Adequate descriptive techniques are not available; these points emphasize some of the problem areas.

6.2.4.1 Function--Architecture Mapping--Though we can use English text to describe the assignment of functional characteristics to physical modules, such informal descriptions do not lend themselves to formal validation or verification techniques. Furthermore, designers are loath to provide any description of this level of their designs.

6.2.4.2 Fault Reconfiguration--Designers do not adequately describe system reconfiguration after a fault has been detected in one of the modules. There are no formal techniques to describe such reconfiguration.

6.2.4.3 Permissible Parallelism--Just as the function specifier should specify all possibilities that he knows for parallel processing (Section 6.2.1.4), so too must the designer specify all parallelism possibilities inherent in his structure. Implementers will have a better chance of finding an efficient implementation if they know all of these options.

6.2.4.4 Synchronization Problems--No designer specified any information about synchronization requirements when the function was implemented on the architectures they designed. We need a systematic technique to capture all synchronization needs. Perhaps software synchronization specification techniques can be adapted to architectural module synchronization specifications.

6.2.4.5 Systems Control Algorithms--Some designers did not specify anything concerning the system control algorithms required to manage the implementation of the function on the new architecture. Those who did were forced to resort to English text. We need a technique to specify these algorithms. Also, we would like to be able to estimate the processing and memory requirements of the control algorithms.

6.2.4.6 Fault Detection Algorithms--Our designers often stated how the architecture should respond to a detected fault. They usually did not, however, state how any faults would be detected. Aside from the standard redundancy techniques, designers should specify options to run, for example, diagnostic programs during periods that particular system modules are idle.

SECTION 7

FUTURE NEEDS

This section follows the structure of Section 6.2 in which we separated the topics according to their positions within the system design cycle. In each subsection, we describe both nonexperimental and experimental needs. We also make some suggestions concerning techniques that might bear the most fruit in meeting these needs.

In Section 7.1, we deal with function specification techniques; in Section 7.2 we discuss design constraints. The designer needs these inputs to effectively perform the design steps. We discuss ways to improve design steps in Section 7.3. Finally, in Section 7.4, we point out how the documentation covering the design could be improved. This documentation could be used to initiate a similar design cycle in which the modules within the architecture are developed.

7.1 FUNCTION SPECIFICATIONS

The following subsections describe the need for further research and development activities to support more accurate, precise, and concise function specifications. Much of this need directly relates to the conclusions described in Section 6.2.1.

7.1.1 Implementation-Independent Specifications

We need specification techniques that do not imply restricted implementations of the function. In fact, we want to leave maximum flexibility for the designer. Most existing precise specifications are tied to implementations or reflect certain implementation decisions because they describe sequential uniprocessing of the function. Data flow techniques[35-37] show sequencing options but do not show the processing details well. C-graphs[38] exhibit data dependencies but are very detailed. Future efforts should build upon existing techniques.

Important parameters characterizing the function should be specified both by mean values and by information concerning the distribution of their values. Techniques to handle probability distribution information must be developed.

7.1.2 Object Interaction Specification

When a function is to be performed on all members of a set of objects, that functional specification can be factored into two parts: the first describes the processing associated with a single object, and the second describes the interactions among the objects. Extant interaction specification techniques include conventional DO loops expressing sequential processing, and parallel DO loops expressing simultaneous processing. These techniques do not allow one to describe simultaneous processing when some interactions occur within the processing sequence. One wants to be able to describe any of these options without biasing the designers. Furthermore, one wants the designer to be able to state whether pipelined processing may be used.

We need to develop specification techniques to encompass pipelined processing and limited-interaction parallel (array) processing without the burden of dividing the task into separate (software-type) tasks which may introduce much unnecessary overhead.

7.1.3 Timing Specifications

Real-time systems designers require many timing specifications. Present techniques include giving deadlines, which are simply maximum permissible values of the intervals elapsing between stimulus and response, rates, which can be converted to time intervals and, thus, to deadlines, and priority schemes, which specify a scheduling algorithm but do not relate directly to timing (since the processing rates are not otherwise specified).

We must develop precise ways to specify deadlines and priorities without implementing a system control algorithm that schedules the resources to meet timing requirements. When selecting a timing specification technique, we should consider compatibility with analytic and simulation tools that might be employed to analyze system performance. We also might develop automated tools to transform timing specifications into simulation programs.

Degradation and saturation timing specifications are also required (see Sections 7.1.4 and 7.1.5).

Timing specifications should not include information concerning the internal behavior of the function when a new input arrives (stating that other processing must be interrupted, for example). The timing specification may, however, include maximum response times for various input classes; a designer should decide how to meet the response time requirement and he may have to use interruptions to achieve that goal.

The probabilistic behavior of the environment, especially the occurrence of input signals, must be specified. We need effective ways to incorporate this information into simulations and other analyses of system behavior. If possible, one should develop automatic methods to translate from input behavior specifications to simulation programs.

7.1.4 Saturation Specifications

Every function specifier should describe the behavior of every function under saturation conditions. Methods to specify the trade between delaying the response and dropping the request must be developed. As a first approach to the problem, we would try specifying a curve indicating the "value" of the response as a function of the delay in receiving the response. This method seems incomplete since it does not include information concerning the other requests competing for processing. Scheduling policies that maximize the cumulative "value" of the responses would, however, account for the competition since high priority requests should have higher "value" associated with their responses.

The problem of maximizing the cumulative value of the responses is similar to the scheduling problem, especially to designer A's approach to that problem (Section 3.3.1).

Another, more complex option to handle saturation is to provide degraded service to some requests, as described in the next section. This technique requires more elaborate response value specifications.

7.1.5 Degradation Specifications

Function specifiers must state whether the system may respond to either saturation or component degradation by providing degraded service to input requests. Degraded services in a discrimination function may include, for example, relaxed tests or minimal signal filtering in place of complex tests.

One technique for specifying degraded performance is to give a set of functions producing different responses (of varying "quality") for the same request, attaching a separate "value" curve to each response. Scheduling the system to maximize the cumulative response value is very complex. Experiments to evaluate the effects of such options would be useful to improve designers' intuitions regarding the value functions. Optimum (but non-realizable) scheduling algorithms could be used in the experiments.

7.1.6 Performance Specifications

Function specifiers must provide information that limits the designer from expanding the system at any cost to provide maximum performance (that is, providing the maximum possible "value," as discussed in Sections 7.1.4 and 7.1.5). Curves specifying permissible trades between cost and value seem the best approach to this problem.

Simulations to study the nature of these trades should be undertaken in a BMD environment so that the system designers can understand the nature of the cost-performance trades for each candidate architecture family. One class of architecture may be optimum for certain parameter ranges, while others would be optimum in other ranges. Present analytic knowledge is not sufficient to study these problems.

7.1.7 Specification Management

A standard form should be developed for function specification. This form must be designed to guide specifiers to provide all information needed by the system designers. We could begin this effort with specifications developed in software engineering studies,[39] modifying them to provide for response time-value curves, control algorithm specifications, and so forth. The information discussed in Sections 6.2.1 and 7.1 should all have a position in the standard format.

An automated design tool could require the problem specifier to provide all this information. A guided tour through a standard question sequence would be a first step in this direction. But first we must develop an explicit list of the information required for the specification.

7.1.8 Machine-Processable Specifications

All specifications should be structured so that they can be processed by machine. Eventually, automated design aids should be developed. Other suggestions regarding machine aids to designers should be considered when the form of the machine-processable requirements is developed.

7.2 DESIGN CONSTRAINTS

In the following subsections, we outline work needed to support more useful specifications of the constraints imposed upon the designer. Such constraints are necessary to prevent the designer from making arbitrary decisions. Most of these suggestions derive directly from our observations discussed in Section 6.2.2.

7.2.1 Payoff Functions

Many payoff dimensions have been identified in this study (see Volume II). We need ways to specify to designers which payoffs should be emphasized in which stage of the design. Performance specifications (discussed in Section 7.1) are only one component in a complex payoff space. Weighting coefficients and an evaluation function must be specified to tell the designer how to select among architectural alternatives.

7.2.2 Component Specifications

Designers must work with a known repertoire of component elements. We need to develop characterizations of module performance that will be useful to distributed system designers. Gross figures, such as MIPS, seem too simple to be adequate for this task. Exercises in which designers are given candidate component specifications would be helpful to reveal how designers use component specifications. Such exercises would guide us toward more useful forms of such specifications.

7.2.3 Component Degradation

Designers must know how well their system components will perform. The only techniques that specify degraded performance are "all-or-none" specifications stating failure probabilities. At higher levels of design, such as the processor-memory-switch level used by our designers, components may degrade without experiencing complete failure. We must develop techniques to describe this behavior.

If a module's performance can be characterized by a small number of parameters, the degradation specification could be a multi-dimensional probability density function stating the probability that the module will exhibit a certain performance vector.

Techniques must be developed to combine component degradation specifications to form system degradation specifications. Experiments may be needed to develop intuition about the combinations although analytic techniques seem most promising to begin this effort.

7.3 DESIGN STEPS

Since our emphasis in the studies reported in this volume has been on the design steps themselves, so our emphasis in suggesting further research and development also lies in the design steps. Research and development concerning design steps can be supported by design exercises but not by experiments. These suggestions derive from the conclusions discussed in Section 6.2.3.

7.3.1 Methodology Development

Our two methodologies differ in that one (Section 2.2) emphasizes spreading the performance of a subfunction across modules within the architecture, while the other (Section 2.1) concentrates subfunctions into closely connected sets of modules. The first, however, does not help the designer find good architectures to implement the subfunction (it simply states "find an architecture to implement the subfunction"). By using the other methodology to subdivide the subfunction, the designer should be able to produce an appropriate distributed architecture to implement the subfunction.

Further refinements of the distributed architecture development step should be made to include additional guidance to the designer. Growth and modularity issues should be intimately involved in these design decisions.

7.3.2 Decomposition Termination Criteria

Our architectural development methodologies, like many development techniques, require decomposition of the desired function as a prelude to synthesis. There are no criteria to guide the designer as he decides whether to terminate decomposition and begin synthesis. Rough guidelines--such as the guideline that any functional module which is too large for a realizable module (in current technology) must be divided further--are not specific enough.

Studies to determine the "values" of decompositions, ascertain the consequences of particular decompositions, and provide termination criteria are needed.

7.3.3 Architecture Evaluation Criteria

Designers who are developing architectures need systematic procedures to evaluate prospective designs against selected performance/payoff criteria. Future work must include the study and development of such evaluation techniques. Even crude evaluations can be useful in eliminating candidate architectures. With crude evaluations, one should provide some estimates of the range of uncertainty in order to know whether there is any reason to keep a design even though its rough evaluation indicates that it should be discarded.

If evaluation techniques could be automated, much of the design process could be automated because the clearly inappropriate options would be discarded at the earliest possible moment.

7.3.4 Verification and Validation

Many real-time systems, including BMD systems, cannot be adequately tested because the tests may have dire consequences. Therefore, it is essential that the design techniques be developed such that the design steps and the designs themselves can be checked against the system requirements. One wants to be able to demonstrate that the requirements will be met if the methodology is followed. We are far from this goal at the present time; further research and development is urgently needed on this issue.

Automated techniques to achieve this goal are far beyond the present state of knowledge.

7.3.5 Memory-Based Designs

The two functions studied during this project emphasized processing requirements; many BMD functions are, in fact, processor bound. There are, however, other functions requiring data base maintenance (track file maintenance, for example) that do not involve much processing. Future research and development should focus on (different) design methodologies for functions whose requirements are dominated by memory needs. Many of the future needs discussed here are relevant to memory-based designs although the words used in this document may emphasize processor-based designs.

Design methodologies for functions dominated by communications requirements should also be studied.

7.3.6 Payoff Emphases

As a designer performs a design step, he usually emphasizes one payoff over the others. This observation raises several problems. First, how should the payoffs be selected for emphasis at each step? Second, should the methodology be changed to include an iterative loop in which different payoffs are emphasized on different passes through the loop? Third, what techniques can the designer use to satisfy each important payoff? Finally, how will the designer document these aspects of his design process? These questions suggest four interrelated future research efforts: payoff selection techniques, methodology changes, payoff realization techniques, and design decision documentation.

Payoff selection techniques must rely on the weightings given by the problem specifier to the designer (see Section 7.2.1). If the designer has not been directed concerning payoff emphasis, he might begin by considering those payoffs which have the greatest disparity between the system requirements and the permissible component specifications. A ratio measure seems best for the first cut at the selection criterion.

We must develop new methodologies incorporating design techniques that allow the designer to refine a preliminary design to enhance its characteristics with respect to a specified payoff. It is not clear how to approach this task.

Third, we should <u>develop a taxonomy of design techniques</u> oriented towards payoff satisfaction. A structure imposed on existing techniques can suggest new techniques and also guide the designer in his creative efforts. The taxonomy could be incorporated within the data base associated with an automated design aid. The payoff consequences of each choice should be associated with the taxonomy whenever possible.

Finally, just as we need to develop a standardized function/design problem specification technique, we also need to <u>develop a standardized format</u> for designers to document their designs, their design steps, and their payoff emphases. If the designer were assisted by an automated mechanism, that system could automatically produce this documentation (by asking direct questions of the designer when necessary).

7.3.7 <u>Relationships Among Payoffs</u>

Various design techniques produce positive contributions to some payoffs and negative contributions to other payoffs. If we could characterize these tradeoffs among the payoffs, we could advise designers on how to select design techniques to implement functions. The presence of positive and negative signs in an interaction matrix (see Volume II) is not adequate; quantified data must be used. Experiments to discover points on the tradeoff curves would be useful.

7.3.8 <u>Automated Aids to Design</u>

As we understand more of the design process, we can automate more of that process. We do not expect to be able to completely automate design, but we feel that appropriate design aids can guide the designer and advise him regarding options. Future efforts should incorporate automated design aids as soon as enough is understood to automate any steps. Then succeeding research efforts would be free to explore other aspects of the design process while holding the known steps fixed (since they are automatically performed).

7.4 ARCHITECTURAL DESCRIPTION TECHNIQUES

We found that designers had to resort to English text to describe the architectures they produced. Diagrams are adequate to convey the hardware structure, and other types of diagrams and listings can convey the functional structure; however, we do not have any technique to show how the two structures are interrelated. We describe several future research areas related to this general problem in the following subsections.

A further aspect of the designer's output is a description of the assumed characteristics of each module used to construct the system. This information should be correlated with the two structures and the description of their interrelationships.

The original design problem was specified as a function to be implemented by an architecture; its result is an architecture whose modules each have been assigned a subfunction to be performed. Each module's specification can then become another design problem at the next lower level of detail. Thus, some of the specification problems described in Section 7.1 apply here. In this section, we discuss only the issues involved in specifying the distribution of functions across modules and the attendant coordination and synchronization problems. These suggestions are developed from the observations discussed in Section 6.2.4.

7.4.1 Function-Architecture Mapping

Suppose that we start with one diagram of the hardware module structure of a system and another diagram of the software module structure. We need to develop a topological mapping to relate the two structures.

One descriptive technique is an assignment matrix A in which A_{ij} is one if and only if software module i is assigned to hardware module j; otherwise A_{ij} is zero. When the function is described for a set of n objects but each software functional diagram is constructed for a single object, the rows of A may contain several non-zero entries. The non-zero entries reflect the number of objects assigned to each hardware module. The row-sums of the entries in A must not exceed the number of objects being handled.

This preliminary descriptive technique requires expansion to handle several complexities. The software may be described as a hierarchy of coordinated flowcharts, as discussed in Section 7.1.2. Similarly, the hardware may be described as a hierarchical structure. These problems complicate module identification techniques.

We need to develop processing techniques to handle these descriptions. We need to consider alternate descriptive techniques that may be more graphic for human understanding or that may be more amenable to machine processing.

7.4.2 System Control Algorithm Development

Given a function, an architecture, and a description of the mapping between the two, we want to find the system control algorithms needed to manage this implementation. It would seem that one could perform this task starting from descriptions of the connectivity of the function description, the connectivity of the hardware description, and the matrix assigning functions to modules. This omits one problem--the memory requirements again have been ignored! Even without memory requirements, we should be able to develop scheduling and synchronization needs automatically from the three descriptions. Further research and development is required to refine these rough ideas.

If the designer provides information concerning the execution times of various functions on the architectural modules, one should be able algorithmically to generate schedules for module usage.

7.4.3 Communications Requirements

We need to develop automatic methods to determine the communications requirements implied by a given function-architecture mapping. The mapping matrix A could be used along with a data-flow description of the function to automatically derive communications requirements. These requirements can be mapped to link bandwidth needs, given the topology of link interconnections.

A related issue concerns the rerouting of communications in the presence of link failure. Efficient techniques to perform this rerouting in the presence of massive link failures are unknown. Statically defined rerouting maps cannot be used because the number of static maps is equal to the number of possible combinations of link failures. This rerouting issue is of paramount importance for communications among the nodes of a BMD system, but it is of small importance within a node since one can assume that either the whole node fails or only a very small number of links may fail. Reconfiguration under bus failure is an unsolved problem.

7.4.4 Fault Reconfiguration

Fault reconfiguration specifications need to be developed. A possible technique would be to define function-architecture mapping matrices A for each possible fault. This task is overwhelming due to the number of modules and, therefore, the large possible number of multiple failure situations.

7.4.5 Fault Detection Algorithms

If module usage schedules can be automatically generated, then one can consider augmenting the automated mechanism to have it schedule the modules for fault detection algorithm execution. Further study is required to evaluate various fault detection algorithms with respect to cost, speed, and their abilities to detect important (for the application) faults. We need to develop techniques to execute fault detection algorithms without destroying the system state for further execution of the application algorithm. This requirement may imply a need for dynamic memory relocation since the memory storing the code cannot be checked without removing the code to other locations.

REFERENCES

1. Gouda, M.G., Han, L.W., Jensen, E.D., Johnson, W., and Kain, R.Y., "Towards a Methodology for Distributed Computer System Design," to be presented at the Sixth Texas Conference on Computer Systems, November 1977.

2. Baer, J.L. and Estrin, G., "Bounds for Maximum Parallelism in a Bilogic Graph Model of Computations," _IEEE Trans._, _C-18_, 11, 1012-1014, November 1969.

3. Boebert, W.E., "Development and Evaluation of the Algorithm Specification," Final Report, Contract No. N00123-76-C-1128, Naval Ocean Systems Center, January 1977.

4. Dennis, J.B., "First Version of a Data Flow Procedure Language," MIT Project MAC Memo TM-61, May 1975.

5. Dennis, J.B. and Fosseen, J.B., "Introduction to Data Flow Schemas," Project MAC, Computation Structures Group Memo 81-1, September 1973.

6. Estrin, G. and Turn, R., "Automatic Assignment of Computations in a Variable Structure Computer System," _IEEE Trans._, December 1963.

7. Martin, D.F. and Estrin, G., "Models of Computational Systems--Cyclic to Acyclic Graph Transformations," _IEEE Trans._, _EC-16_, 2, 70-79, February 1967.

8. Project MAC Conference on Concurrent Systems and Parallel Computation, June 1970.

9. Rodriguez, J.E., "A Graph Model for Parallel Computation," PhD Thesis, Electrical Engineering Department, MIT, September 1967; also on MIT Project MAC Report MAC-TR-64.

10. Slutz, D.R., "The Flow Graph Schemata Model of Parallel Computation," PhD Thesis, Electrical Engineering Department, MIT, 1968.

11. Chandy, K. M. and Reynolds, P. F., "Scheduling Partially Ordered Tasks with Probabilistic Execution Times," ACM Operating Systems Review, 9, 5, November 1975, pp. 169-177.

12. Coffman, E. G., Jr. and Graham, R. L., "Optimal Scheduling for Tw Processor Systems," Acta Informatica, 1, 1972, pp. 200-213.

13. Karp, R. M., "Reducibility Among Combinatorial Problems," Complexity of Computer Computations, Miller and Thatcher, eds., Plenum Press, 1972, pp. 85-104.

14. Karp, R. M. and Miller, R. E., "Properties of a Model for Parallel Computations: Determinancy, Termination, Queuing," SAM J. Appl. Math, 11, 6, November 1966.

15. Sethi, R., "Scheduling Graphs of Two Processors," SIAM J. on Computing, 5, 1, March 1976, pp. 73-82.

16. Ullman, J. D., "Polynomial Completeness of the Equal Execution Times Scheduling Problems," Department of Electrical Engineering, Princeton University, Computer Science Report TR-115, December 1972.

17. Holt, R. C., "Some Deadlock Properties of Computer Systems," ACM Computing Surveys, 4, 3, September 1972, pp. 179-196.

18. Karp and Miller, op. cit.

19. Lien, Y. E., "Termination Properties of Generalized Petri Nets," SIAM J. on Computing, 5, 2, June 1976, pp. 251-265.

20. Ross, D. T. and Schoman, K. E., Jr., "Structural Analysis for Requirements Definition," IEEE Trans., SE-3, 1, January 1977, pp. 6-15.

21. Deley, G.W. and Wisehart, W.R., "Data Processing Subsystem Engineering (DPSIE) Benchmark Experiment," General Research Corporation, CR-3-685, March 1976.

22. General Research Corporation, "Preliminary Hardsite Demonstration Software Specification," CR-1-197/1, January 1971, revised January 1972.

23. Wishner, R.T. and Hastings, F.A., "Camel Radar Scheduling Processing Study," RADC-TR-72-1, October 1971, (Secret) pp. 19-33.

24. Evensen, A.J. and Troy, J.L., "Introduction to the Architecture of a 288-element PEPE," Proceedings 1973 Sapamore Conference on Parallel Processing, pp. 162-169.

25. Flynn, M.J., "Some Computer Organizations and Their Effectiveness," IEEE Trans., C-21, September 1972, pp. 941-960.

26. Ullman, J.D., op. cit.

27. Ramamoorthy, C.V. and Li, H.F., "Pipeline Architecture," ACM Computing Surveys, 9, 1, March 1977, pp. 61-102.

28. Enslow, P.H., Jr., "Multiprocessor Organization--A Survey," ACM Computing Surveys, 9, 1, March 1977, pp. 103-129.

29. Arnold, R.G., private communication, June 1977.

30. Gouda, M.G., "Protocol Machines: Toward a Logical Theory of Communication Protocols," Ph.D Thesis, Computer Science, University of Waterloo, 1977.

31. Gouda, M.G. and Manning, E.G., "Probabilistic Cost Machines," Symposium on New Directions and Recent Results in Algorithms and Complexity, Carnegie-Mellon University, April 1976.

REFERENCES (Concluded)

32. Carter, W.C. and Schneider, P.R., "Design of Dynamically Checked Computers," Information Processing 68, North Holland Publishing Company, pp. 878-883.

33. Wakerly, J.F., Low-Cost Error Detection Techniques for Small Computers, Ph.D Thesis, Stanford University, 1973.

34. Wakerly, J.F. and McClusky, E.J., "Design of Low-Cost General Purpose Self Checking Computers," Information Processing 74, North Holland Publishing Company, pp. 108-111.

35. Dennis, op. cit.

35. Dennis and Fosseen, op. cit.

37. Dennis, J.B., Misunas, D.P., and Thiagarajan, P.S., "Data-Flow Computer Architecture," Project MAC, Computations Structures Group Memo 104, August 1974.

38. Boebert, op. cit.

39. Ross, op. cit.

BIBLIOGRAPHY

Barnes, G.H., et. al., "The ILLIAC IV Computer," IEEE Trans., C-17, 8, August 1968, pp. 746-757.

Batcher, K.E., "STARAN Parallel Processor System Hardware," NCC, 1974, pp. 405-410.

Berg, R.O., et. al., "PEPE--An Overview of Architecture, Operations, and Implementation," Proceedings of National Electronics Conference, 1972, pp. 312-317.

Parnas, D.L., "A Technique for Software Module Specification with Examples," Comm. ACM, 13, 5, May 1972.

Parnas, D.L., "The Use of Precise Specifications in the Development of Software," Information Processing 77, edited by B. Gilchrist, North Holland Publishing Company, 1977, pp. 861-867.

Watson, W.J., "The TIASC--A Highly Modular and Flexible Super Computer Architecture," AFIPS Conference Proceedings, 41, FJCC 1972, pp. 221-228.